IN THE UNITED STATES PATENT AND TRADEMARK OFFICE #4

In re Patent Application of

BOYLAN et al.

Atty. Ref.: 922-143

Serial No. 09/919,806

Group: 2825

Filed: August 2, 2001

Examiner:

For: AUTOMATIC GENERATION OF INTERCONNECT COMPONENTS

* * * * * * * * * * *

September 25, 2001

Assistant Commissioner for Patents
Washington, DC 20231

SUBMISSION OF PRIORITY DOCUMENTS

Sir:

It is respectfully requested that this application be given the benefit of the foreign filing date under the provisions of 35 U.S.C. §119 of the following, a certified copy of which is submitted herewith:

Application No.	Country of Origin	Filed
0104945.1	Great Britain	28 February 2001

Respectfully submitted,

NIXON & VANDERHYE P.C.

By: Larry S. Nixon

Larry S. Nixon
Reg. No. 25,640

LSN:vc
1100 North Glebe Road, 8th Floor
Arlington, VA 22201-4714
Telephone: (703) 816-4000
Facsimile: (703) 816-4100

U.S.S.N 09/919,806

INVESTOR IN PEOPLE

The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.

Signed M Cooke

Dated 13 August 2001

28 February 2001

28FEB01 E609756-1 D02536
P01/7700 0.00-0104945.1

Patents Form No. 1/77

The Comptroller
The Patent Office
Cardiff Road
Newport
Gwent NP10 8QQ

REQUEST FOR THE GRANT OF A PATENT

The grant of a patent is requested by the undersigned on the basis of the present application:

1 Title of the invention: **Automatic generation of interconnect logic components**

2. Applicants details:

First or only applicant: **3Com Corporation**

Country: USA State: Delaware

Address: 5400 Bayfront Plaza
Santa Clara
California 95052-8145
United States of America

0469100090001

3. Name of agent. **Bowles Horton** ADP Code: **8805003**

Agent's address: Felden House, Dower Mews
High Street, Berkhamsted
Hertfordshire HP4 2BL

4. Agent's reference: **105362**

5. The application claims an earlier date under any of Sections 8(3). 12(6). 15(4) or 37(4): **NO**

6. Declaration of Priority (if any)

Country	Application Number	Priority date
None		

28 February 2001

Patents Form 1/77 (continued)

7. Inventorship

The applicant(s) is/are the sole inventor or joint inventors: **NO**

8. Check List of Documents:

A. The application is accompanied by the following number of sheets:

Request:	**2**
Description:	**54**
Claims.	**1**
Abstract.	**1**
Drawings	**11**

B The application as filed is accompanied by the following:

Patents Form 7/77.	**NO**
Patents Form 9/77:	**NO**
Patents Form 10/77:	**NO**
Priority Document:	**NO**
Translation of priority document:	**NO**

9 Signature ---------- for BOWLES HORTON

0000407 28-Feb-01 04:14

AUTOMATIC GENERATION OF INTERCONNECT LOGIC COMPONENTS

This invention relates to the generation of large scale integrated circuits and particularly to the layout of a 'system on a chip'.

There are various program tools used in the generation of large scale integrated circuits that use libraries of re-useable elements; examples are layout tools with memory libraries. In the case of these tools one still has to hand-code how the individual elements are connected together. A new design using the same set of libraries elements but a different interconnect hierarchy or architecture requires the designer to hand code this interconnect logic afresh

The present invention partly relies on a library of reusable elements but automates the generation of the interconnect logic. This permits automatic generation of new and different realisations of the architecture.

The preferred architecture means that substantially all data exchange between core blocks is via a central shared memory (or group of memories) that could be on-chip and/or off-chip. This means that if for example an Ethernet Core and PCI core have to pass data to each other then the data would be copied into memory from and by the Ethernet core and copied out of memory by the PCI core

Access to memory is a limited resource. Preferably therefore the invention includes a hierarchical data aggregation technique. Thus cores must go through successive levels of arbitration in other to gain access to memory. This has two main advantages in that it allows dispersal of routing bottlenecks and enables the use of the lowest possible frequency clocking for each operational function.

Preferably there is a separation of the data path from the register path. Data handling cores communicate with memory via the data path. Register paths are between processor cores and other cores. It is possible to have multiple register paths from processor cores to

groups of cores. This allows us to group cores on a particular register path based on such parameters as bandwidth and latency

In the accompanying drawings:

Figure 1 is a data path diagram

Figure 2 is a register path diagram

Figure 3 is another register path diagram.

Figure 4 is another data path diagram

Figure 5 is a further data path diagram.

Figure 6 is a table of states for a state machine.

Figure 7 is a timing diagram for the state machine shown in Figure 6.

Figure 8 is a diagram illustrating an interconnection hierarchy.

Figure 9 is a diagram illustrating high level clock functions.

Figure 10 is a diagram illustrating bridge logic.

Figure 11 is a diagram illustrating arbitration functions.

Figure 12 is a further diagram illustrating arbitration functions.

Figure 13 is a diagram illustrating bus paths.

Figure 14 is a diagram illustrating wrapper functions.

For N>2 (N=number of devices interconnecting), in an SoC or similar application, a scheme according to the invention infers automatically appropriate logic functions, such as arbitrators, inter-clock-domain boundary buffering and alignment, clocking mechanisms. Interconnections may be depicted graphically or other wise.

The key to developing systems quickly is the separation of the interconnect logic and the basic operational blocks, herein called 'cores'. These cores will not need to be altered for each system; only the set of cores and the way they interconnect need change. The following description describes how the generation of this interconnect logic (which is preferably expressed in HDL/ Verilog) can be automated.

The inputs needed to automatically generate the interconnect logic are as follows:

1. A Library of reuseable cores with key parameters defined from which cores can be selected:

2. A set of rules defining how cores can be connected together:

3. Interconnect logic blocks (Clock Generator, Arbitration, Register Bridge) and their configurable parameters:

4. A method of describing how the cores need to be connected. This could be achieved using a spreadsheet or as preferred a graphical picture showing the cores and how they are connected together, as shown for example in Figure 1:

5. Generic Verilog for each of the Interconnect blocks to which the parameters can be applied to create the specific interconnect logic for the system being designed.

Using these inputs a set of algorithms will be applied in order to create the system's interconnect logic. There are effectively three generic types of algorithms that can be applied in order to create the logic.

(A) Parameterisable Verilog - where all that needs to be done is to define the value of a set of parameters

(B) Verilog Templates - templates are used where the same functionality needs to be repeated a number of times. Examples are generation of select line logic (select 1 of N blocks connected via the same Bus) or multiple instances of the same interface logic (e.g. an Arbitration Block with 5 mBus connections).

(C) State Machine Algorithms - All the verilog is generated. The algorithm decides the number of states in the state machine and the value of all signals in the state machine.

In order to generate the logic associated with a particular interconnect block it may be necessary to apply combinations of these algorithms one or more times. The Verilog or HDL modules will be created for each of the interconnect blocks shown graphically in the interconnect diagrams or otherwise. A top level verilog instantiation file with be created incorporating each of the interconnect blocks and core wrappers. This file will declare an instance for the generated modules (Arbitration register bridge etc). It will declare an instance for each of the core wrappers. The verilog instantiation file will reflect a completely flat hierarchy with all modules being declared at the same level. This will be the starting point used to create selectable hierarchies.

Rules

The following are the rules for drawing register path diagrams as shown in Figures 2 and 3.

1 One may add any core that has an rBus target interface.

2 One may add any number of cores

3 A core may appear on one rBus only

4 Cores that do not require to use rBus Target functionality should be placed on a 'null register' bridge. This will ensure that unused input signals will be tied off.

5 Cores may have only one rBus Target port.

6 A Register Bridge may have any number of mBus Target ports.

7 A Register Bridge may have only one rBus Initiator Port.

8 All cores connected to a register bridge must have a clock frequency greater than or equal to the Register Bridge's clock frequency.

Rules

1. One may have only the following elements in a data diagram:

Core	Register Bridge
mBus Initiator Port	Arbitrator
mBus Target Port	Clock Generator

2. One must have at least 1 Core with an Initiator port and 1 Core with a Target port
3. One may have only a maximum of 64 Cores (limited by the Source Identifier size – 6 bits).
4. One may have multiple instances of the same core (MAC1, MAC2 etc).
5. One may never have a core with both an mBus Initiator and mBus Target interface.
6. One may have any number of Arbitration, Register Bridge and Clock Generator blocks.
7. Initiator connects to Target(s) / Register Bridge(s) / Arbitrator(s).
8. Target connected to by Initiator (1) / Arbitrator (1).
9. Register Bridge connected to by Initiator(s) / Arbitrator(s).
10. Arbitrator connected to by Initiator(s) / Arbitrator(s).
11. Arbitrator connects to Arbitrator(s) / Target(s) / Register Bridge(s).
12. Only Processor Cores should be able to address Register Bridges. There should not be any paths through the interconnect hierarchy that allows a non-processor core connect to a Register Bridge.
13. There should only be one possible unique path from an mBus Initiator interface to an mBus Target interface.
14. Cores may have multiple mBus Initiators interfaces (maximum number is a Library property section 6)
15. A Memory Interface Core may have only one mBus Target.

16 An mBus Interfaces on a core that is unused will automatically have its unused Input signals tied off

17 An mBus may be split so that it goes to multiple blocks The maximum number of destinations is a Library property for Cores and is configurable for Arbitration blocks.

18 An Arbitration block may have any number of input ports but may have only one output port.

An example of a data path diagram without clock generation blocks is shown in Figure 4.

All blocks (cores, register bridges, arbitrators) in the data diagram must be connected to a clock generator block. Blocks that run at the same clock frequency can be connected to a common clock generator block. If any block in a group of blocks connected to a clock generator wishes to use a logic clock then all blocks in the group must use a logic clock. A group of blocks that have a common clock generator will need to be physically close to each other in the final layout of the ASIC. Clock generator blocks derive their required clock frequency from the system clock unless specifically connected to another 'parent' clock generator. in which case their clock frequency is derived from the parent blocks clock frequency.

Figure 5 is a data path diagram including clock generator blocks.

A clock generator can be used as a parent if (a) none of the blocks below it in the interconnect hierarchy talk directly to any other blocks at a higher level and if (b) all blocks below it in the interconnect hierarchy can have their clock frequency derived from it.

The following abbreviations are used to specify parameter behaviour:

- **RW** - The parameter can be read and written to by the tool.
- **RI** – The value of the parameter is inferred from one of the interconnect diagrams. The value of the parameter can be read.
- **I** – The value of the parameter is inferred from the one of the interconnect diagrams
- **R** - The value of the parameter can be read and its value is taken from the core library.

The parameters define what is configurable. They do not place any restrictions on how the parameterised verilog is created

Table 1 below shows examples of global parameters by name. value, type and description. Table 2 and Table 3 similarly show the parameters for a clock generator block.

TABLE 1

Parameter Name	Value	Type	Description
System_Clock	Integer	RW	This is the master clock that is sent around the chip to create lower frequencies clocks. The System clock will have a default value of 200.
Max_Burst_Size	Integer	RW	The maximum burst size (read or write) which is allowed on the mBus. The default maximum burst size will be 32.

TABLE 2

Parameter Name	Value	Type	Description
Block_Name	String	RW	All blocks in the diagram must have a unique name. The block name will be used in the generation of verilog signal names etc. The default name will be 'Clk#'.
Parent_Clock	Integer	RI	This will be either the system clock or the output from another clock generator block. The Clock_Frequency used by connected blocks will be generated from it.
Is_Logic_Block	Boolean	RW	If this parameter is true then a Logic_Clock will be generated and must be used by all connected blocks. The default value is False.
Clock_Frequncy	Interger	RW	The clock frequency at which the logic of Blocks connected to this Clock Generator will run at. The System_Clock must be an integer multiple of this value.

The design tool will traverse the data diagram to create an array of divide by numbers for each lower frequency block connected to the set of blocks for which this clock generator is

generating a clock frequency. The 'divide by' ratio array will be used in the generation of Sample and Strobe signals The parameter is shown in Table 3.

TABLE 3

Parameter Name	Value	Type	Description
Divide_By_Ratio [n]	Integer Array	I	The divide by ratio of each connected block of lower frequency. The divide by ratio is calculated by dividing the Parent_Clock value by the Clock_Frequency of the lower frequency block.

Table 4 illustrates the parameters for an arbitration block

TABLE 4

Parameter Name	Value	Type	Description
Block_Name	String	RW	All blocks in the diagram must have a unique name. The block name will be used in the generation of verilog signal names etc. The default name will be 'Arb#'.
Clock_Frequency	Integer	RI	The clock frequency at which the logic associated with this block will run at. The System_Clock must be an integer multiple of this value.
No_Of_Ports	Integer	I	This value is inferred from the data diagram. Each Arbitration block will initially have 2 input ports and one output port.
SID_To_Port_No [n]	Integer Array	I	An entry in the hash table or array will exist for all cores below this Arbitration block in the interconnect hierarchy. Source Identifiers are used on the return path of the mBus to identify the Source of a read or write request.
Required_Bandwidth	Integer	RW	The bandwidth required by this Arbitration block. The default value for Required_Bandwidth is calculated by summing the allocated bandwidth at each of the arbitration block's input ports

Each mBus Input port in an Arbitration block will have two types of buffers:-

Up_Buffers stores mBus read and write requests going up the interconnect hierarchy towards an mBus Target. The size of some of the Up_Buffers is fixed (rdCmdData and rdCmdPhase) and the size of others is variable (wrInfo, wrPhase, wrData). The minimum size of the variable Up_Buffers is dependant on the Max_Burst_Size.

Down_Buffers stores mBus read responses going down the interconnect hierarchy towards mBus Initiators The size of the Down_Buffers is variable (rdDataPhase, rdData). The minimum size of the variable Down_Buffers is dependant on the Max_Burst_Size.

The relevant parameters are shown in Table 5 below.

TABLE 5

Parameter Name	Value	Type	Description
Bandwidth_Allocation	Integer	RW	The bandwidth to be allocated by the Arbitration block to this mBus input port. The default value will be inferred from the block connected below it in the interconnect hierarchy. The default value will be the lesser of the following two values Required_Bandwidth or Output_Bandwidth.
Min_Buffer_Size	Integer	RI	This is the minimum size of the buffers for this mBus input port. The buffers cannot be decreased below this size. The value is inferred from the diagram by multiplying the Max_Burst_Size by the mBus_Width associated with this port.
Up_Buffer_Size	Integer	RW	The integer number of storage locations in the Up_Buffers. The size of a each storage location is dependant on the specific buffer and the mBus_Width of the port. It can never be less than Min_Buffer_Size which is its default value.
Up_Buffer_Assert	Integer	RW	How full the Up_Buffers needs to be before you can attempt to pass information up the interconnect hierarchy.
Up_Buffer_Accept	Integer	RW	The number of storage locations that need to be available in the Up_Buffers before they will accept new information.

Down Buffer_Size	Integer	RW	The integer number of storage locations in the Down_Buffers. The size of a each storage location is dependant on the specific buffer and the mBus_Width of the port. It can never be less than Min_Buffer_Size which is its default value.
Is Throttled	Boolean	RW	This will stop requests being sent up from the corresponding Up Buffer if the Down Buffer is full. It will default to False in most Arbitration blocks. It will default to True for Arbitration blocks directly connected to mBus Targets.

The paramters for mBus Output Ports are shown in Table 6 below.

TABLE 6

Parameter Name	Value	Type	Description
Output_Bandwidth	Integer	RI	The bandwidth available at this mBus Output port. The value is inferred from the diagram by multiplying the Clock_Frequency of the Arbitration block by the Bus_Width of this mBus.
Bus_Width	Enum	RW	The width of this mBus. The supported values are 8.16.32 and 64. The default value will be 32.
Duplex_Mode	Enum	RW	The duplex mode of the mBus. The supported values are Half, Full. The default value will be Half.
Addressable_Targets [n][3]	2-Dim Integer Array	I	This array or hash table will define the upper 16 bits of the Base_Address of all mBus Targets reachable through this output port. It will also define the offset (in 64K increments) to the end of the memory associated with each mBus Target. (see section 7 for more details). It will identify the input port of the higher level block through which the mBus Target is accessible (mBus can be split to multiple destinations).

The parameter for a rBus Half Duplex Target Port is shown in Table 7 below.

TABLE 7

Parameter Name	Value	Type	Description
Address_Bits_Decoded	Integer	RI	The number of address bits decoded defines the number of registers in this Arbitration block. The value is inferred from the diagram see section 7 for more details on how it is calculated.

Is Throttled will be turned on by default in any Arbitrator connected to an mBus target (Memory or register bridge): it will be turned off by default in all other arbitration. Arbitration blocks directly connected to a memory interface preferably have a 64 bit wide mBus.

The parameters for a Register Bridge (with in-built Arbitrator) are shown in Table 8 below.

TABLE 8

Parameter Name	Value	Type	Description
Block_Name	String	RW	All blocks in the diagram must have a unique name. The block name will be used in the generation of verilog signal names etc. The default name will be 'Bridge#'.
Clock_Frequency	Integer	RI	The clock frequency at which the logic associated with this block will run at. The System_Clock must be an integer multiple of this value.
No_Of mBus_Ports	Integer	I	This value is inferred from the data diagram. A Register Bridge will initially have 1 mBus Target port.
Base_Address	Integer	RW	The base address of this mBus Target. See section 7 for details of how it is assigned.

The parameter for a rBus Half Duplex Initiator Port is shown in Table 9.

TABLE 9

Parameter Name	Value	Type	Description
Bus_Width	Enum	RW	The width of this rBus. The supported values are 8,16,32 and 64. The default value will be 32. The same rBus will be feed to all rBus Targets connected to this Register Bridge.

For each of the rBus Targets connected to this Register Bridge one will store the parameters shown in Table 10.

TABLE 10

Parameter Name	Value	Type	Description
Start_Address_Offset	Integer	I	Used in the selection of an rBus Target connected to this register bridge. Each rBus Target has a sequential range of valid addresses. The start address of this range is calculated by adding the Start_Address_Offset to the Base_Address of the Register Bridge See section 7 for more details.
End_Address_Offset	Integer	I	Used in the selection of an rBus Target connected to this register bridge. Each rBus Target has a sequential range of valid addresses. The end address of this range is calculated by adding the End_Address_Offset to the Base_Address of the Register Bridge. See section 7 for more details.

The register bridge arbitration algorithm will preferably be fixed as round robin. This means that it does not require any buffering and that there is no concept of bandwidth allocation on the rBus bus. The rBus will preferably always operate in Half Duplex mode. The total bandwidth on the rBus is defined as (Register Bridge Clock_Frequency * Bus_Width).

The parameters for a core block are shown in Table 11.

TABLE 11

Parameter Name	Value	Type	Description
Block Name	String	RW	All blocks in the diagram must have a unique name. The block name will be used in the generation of verilog signal names etc. The default name will be derived from the core's library property.
Clock_Frequency	Integer	RI	The clock frequency at which the logic associated with this block will run at. The System_Clock must be an integer multiple of this value.
Source_Code_Directory	String	RW	Where the source code for this core is stored. The default value will be taken from the core's library property.
No Of_mBus Ports	Integer	1	Number of mBus ports that this block supports. The ports will all either be mBus initiator ports or mBus Target ports. The value cannot be greater than the library property but mBus ports can be left unused.
mBus Type	Enum	R	Is this core an mBus target or an mBus Initiator. The supported values are Target. Initiator. The value is taken directly from the core's library property.

The parameters for an mBus Initiator are shown in Table 12.

TABLE 12

Parameter Name	Value	Type	Description
Source_Identifier	Integer	RI	This will be a value in the range [0-63]. Source Identifiers are used on the return path of the mBus to identify the Source of a read or write request. mBus Targets will not be allocated a Source Identifier.
Is_Processor	Boolean	R	This value will be set to True if this core is a processor The value is taken directly from the core's library property.

The parameters for mBus Initiator Ports are shown in Table 13.

TABLE 13

Parameter Name	Value	Type	Description
Bus_Width	Enum	RW	The width of this mBus. The supported values are 8,16,32 and 64. The default value will be 32.
Duplex_Mode	Enum	RW	The duplex mode of the mBus. The supported values are Half, Full. The default value will be Half.
Required_Bandwidth	Integer	RW	The bandwidth required by the Core on this mBus Output port. The default value is taken directly from the core's library property.
Output_Bandwidth	Integer	RI	The bandwidth available at this mBus Output port. The value is inferred from the diagram by multiplying the Clock_Frequency for the Core by the Bus_Width of this mBus.
Addressable_Targets [n] [3]	2-Dim Integer Array	I	This array or hash table will define the upper 16 bits of the Base_Address of all mBus Targets reachable through this port. It will also define the offset (in 64K increments) to the end of the memory associated with each mBus Target. (see section 7 for more details). It will identify the input port of the higher level block through which the mBus Target is accessible (mBus can be split to multiple destinations).

The parameters for an mBus Target are shown in Table 14.

TABLE 14

Parameter Name	Value	Type	Description
Base_Address			The base address of this mBus Target. See section 7 for details of how it is assigned. Initiators will not have a Base_Address.
Is_Memory_Interface	Boolean	R	This value will be set to True if this core is a memory interface. The value is taken directly from the core's library property.

The parameters for mBus Target Ports are shown in Table 15.

TABLE 15

Parameter Name	Value	Type	Description
Bus_Width	Enum	RW	The width of this mBus. The supported values are 8.16.32 and 64. The default value will be 64.
Duplex_Mode	Enum	RW	The duplex mode of the mBus. The supported values are Half. Full. The default value will be Half.
Bandwidth_Allocation	Integer	RW	The bandwidth to be allocated by the mBus Target to this mBus input port. The default value is taken directly from the core's library property.
Memory_Bandwidth	Integer	R	The bandwidth available to memory. The default value is taken directly from the core's library property.

The parameters for an rBus Half Duplex Target Port is shown in Table 16.

TABLE 16

Parameter Name	Value	Type	Description
Address_Bits_Decoded	Integer	RI	The number of address bits decoded defines the number of registers in this block. The value is inferred from the diagram see section 7 for more details on how it is calculated.

Signals

The Verilog source code for a core will be interrogated and the following values will be extracted for each of the signal:

- Signal Name
- Signal Width
- Signal Direction
- Is it an External Signal (Pin out)
- Value to tie an Input signal to if it is unused.

There may be an optimum assignment algorithm for the source identifiers to reduce the length of the decode sequence in the Arbitration blocks.

Connections / Bus Paths

It is possible to specify a unique name for all the possible connection on the diagrams. as shown in Table 17

TABLE 17

Connection Type	Default Name
mBus	*'Block Name1'*_*'Block Name2'*_M
rBus	*'Register Bridge name'*_R
Clock Line	*'Clock Frequency'*_Clk
Parent Clock Line	*'Clock Frequency'*_PClk

Table 18 illustrates the properties of each core in a library.

Core Name
Range of Clock Frequencies supported (used to decide if logic clock needed)
Is this an mBus Initiator or an mBus Target
Number of mBus ports (target or initiator)
Is it a processor?
Is it a memory interface?
Is full source code available?
Source code storage area
Description of Core functionality
Estimated Gate Count
Process Geometry supported
Estimated Power Consumption
Core Internal Frequency

Table 19 illustrates the properties which are preferably defined for each mBus Initiator port on the core:

TABLE 19

Bus Widths supported
Duplex Modes supported
Required Bandwidth
Maximum number of selectable mBus destinations

Table 20 illustrates the properties which are preferably defined for each mBus Target port on the core:

TABLE 20

Bus Widths supported
Duplex Modes Supported

Each mBus Target will define one overall bandwidth to memory

The defined property for a rBus Half Duplex Target is preferably the number of bits decoded which defines the number of registers in the core.

The memory map which preferably assumes a fixed address size (e.g. 32 bits) allows the specification of the base address of each block with one or more mBus Target ports. The mBus targets will be extracted from the data diagram. An mBus target may be memory, a register bridge or a mailbox memory. All base addresses should be aligned at a 64K boundary.

Ordinary Memory: Address Pool size

The size of the address pool assigned to normal memory should be configurable. The size of the memory address pool can be incremented in 64K increments.

Register Bridge: Address Pool Size

The size of the address pool assigned to a Register Bridges should be fixed. It should be possible to calculate this fixed value from the register path diagram (I.E. number of rBus Targets connected to the register bridge).

The size of the memory pool assigned to each rBus Target can be calculated as follows:

The bank of registers in any rBus Target will always be assumed to be an integer multiple of 32 (see note 2) Thus if an rBus target has 'n' registers then the size of the memory pool assigned to the rBus target will

F = (n+m) where (n+m) % 32 = 0 and there will be m unused addresses

The pool of memory addresses assigned to a Register Bridge will always be an integer multiple of 64K (see Note 1 below) Then the size of the memory pool assigned to the Register Bridge will be Z = (ΣF)+G where ((ΣF)+G) % 65535 = 0 and there will be G unused addresses

Calculating the number of registers in a block

The number of registers in an core is taken directly from the core's library property.

The number of registers in an arbitration block can be calculated using the formula (OR something similar to this)

x=p(q) where x is the number of internal registers. p is the number of input ports and q is the number of registers at each input port.

All memory needs in this example to be aligned on 64K boundaries because the Arbitration blocks only look at the top 16 bits of an address in order to decide on which path an mBus target is located

The registers in an rBus Target will be an integer multiple of 32 because it means that only the top 11 bits need examination to decide which rBus Target is being addressed.

Register bridge feedback may indicate the amount of register bus bandwidth utilised and the size of memory pool due to number of cores on bus. A warning may be given if the Arbitration blocks required bandwidth is greater than its output bandwidth (freq * bus width).

Warnings may be generated for all unused signals / interfaces in a core. A warning may be generated if one is using unreleased source code in the generation of the interconnect logic. A warning will be generated if the Bandwidth_Required is greater than the "Output_Bandwidth" on any mBus Initiator port. A warning may be generated if the sum of the bandwidths allocated is greater that the "Memory_Bandwidth" of an mBus Target. A warning will be generated if there are no rBus targets on a Register Bridge.

Top Level Functions

The following pseudo-code describes the top level steps used to automatically generate the Interconnect Logic:

CLK_BLK[..] = array of Clock Generator objects of size NO_CLK_BLK

REG_BLK[..] = array of Register Bridge objects of size NO_REG_BLK

ARB_BLK[..] = array of Arbitration objects of size NO_ARB_BLK

IPWRAPPER[..] = array of IP Core Wrapper objects of size NO_IPWRAPPER_BLK

VALID = boolean value used to decide if the interconnect hierarchy that you want to create Logic for if valid.

```
VALID = Validate Interconnect Hierarchy( )
If (VALID == 0)
        Exit

For (n=0) -> (n = NO_CLK_BLK-1)
        Create Clock Logic( CLK_BLK[n] )
        Add to Instantiation File ( CLK_BLK[n] )
For (n=0) -> (n = NO_REG_BLK-1)
        Create Bridge Logic ( REG_BLK[n] )
        Add to Instantiation File ( REG_BLK[n] )
For (n=0) -> (n = NO_ARB_BLK-1)
        Create Arbitration Logic ( ARB_BLK[n] )
        Add to Instantiation File ( ARB_BLK[n] )
For (n=0) -> (n = NO_IPWRAPPER  BLK-1)
        Create IP Wrapper Logic ( IPWRAPPER[N] )
        Add to Instantiation File ( IPWRAPPER[N] )
```

Validate Interconnect Hierarchy

Steps may be included to check if the Interconnect Hierarchy is valid i.e. to check if any architectural assumptions, interconnection rules or clock generation rules are broken.

Creation of Clock Logic

The following pseudo-code describes the steps used to create the Logic for a Clock Generator Block. A verilog module is created with a state machine which generates all the required Clock signals for the connected blocks.

The Clock Generator Object holds the following Information.

NAME = Unique name for this Clock Generator Block

CLK_FREQ = Clock frequency generated by block

PARENT CLK = Parent Clock used to derive the generated Clock frequency

IS_LOGIC_CLK = boolean value which specifies if a Logic Clock should be generated or not.

CLK_SIGNALS[..] = array of objects from the blocks connected to the Clock Generator of size CONNECTIONS. Holds information such as divide by ratios etc.

Clock Edge Identification (CLK_RATIOS[..]) (see section 4 of sdg000.009)

For (n=0) -> (n = CONNECTIONS-1)

Choose CLK divide Function (CLK_RATIO[n].IS_LOGIC_CLK)(section 4.1 sdg000.009)

For (n=0) -> (n = CONNECTIONS-1)

Clk Generation Algorithm (CLK_SIGNALS[n]) (section 5 sdg000.009)

Strobe Signal Algorithm (CLK_SIGNALS[n]) (section 6 sdg000.009)

Sample Signals Algorithm (CLK_SIGNALS[n]) (section 8 sdg000.009)

Create Bridge Logic

The following pseudo-code describes the steps used to create the Logic for a Register Bridge.

The Register Bridge Object holds the following Information

NAME = Unique name for this Register Bridge

CLK_SIGNALS = Reference to an object which describes the Clock Signals for this Block (see section 5)

BASE_ADDR = The Base address for the block

MBUS_PORTS [..] = array of connected mBus Port objects of size NO_MBUS_PORTS

RBUS_WIDTH = the width of the rBus.

RBUS_TARGETS[..] = array of connected rBus Target objects of size NO_RBUS_TARGETS.

Clock Sync (CLK SIGNALS)

For (n=0) -> (n = NO_MBUS PORTS-1)

mBus Target Logic (MBUS_PORTS [n])

Round Robbin Arbitrator (NO_MBUS_PORTS)(see section 6 1 for more details)

mBus to rBus state machine ( )

For (n=0) -> (n = NO_RBUS_TARGETS-1)

rBus Select Logic (RBUS_TARGETS[n])

rBus Initiator Port ( )

Round Robin Arbitrator

The following code describes how the register bridges round robin arbitrator is created. The arbitrator can parameterised to include the required number of requestors (initiators). For half duplex initiators there will be two requests one for read command and one for a write. Therefore each half duplex initiator interface within the register bridge will issue two requests to the arbiter. For a full duplex initiator there will also be two requests but they will be issued from separate read and write interfaces They will be seen as requests from separate initiators.

For 1 to N initiators. where N is 3 .there will be 6 requests to the arbiter for access to the rBus.

module RegisterArb (

For (N = 0) -> (N = No_of_requestors - 1)

add 'request_N. '

done. rst. brClk.

For (N = 0) -> (N = No of_requestors - 1)

```
                add 'space_avail_N.'
                add 'gnt_N.'

            gnt, gntSel);

5

        For (N = 0) -> (N = No_of_requestors -1)
            add 'input request_N; wire request_N;'

        input done;
10      wire done;
        input rst;
        wire rst;
        input brClk;
        wire brClk;

15

        For (N = 0) -> (N = No_of_write_requestors -1)
            add 'input spaceAvail_N, wire spaceAvail_N;'
        For (N = 0) -> (N = No_of_requestors - 1)
            add 'output gnt_N; reg gnt_N;'

20

        output gnt; reg gnt;
        output [1:0] gntSel;
        reg [1:0] gntSel;

25      For (N = 0) -> (N = No_of_requestors -1)
            add 'latchReq_N.'

        reg [1:0] latchSel;
        reg checkDone.

30

        For (N = 0) -> (N = No_of_requestors -1)
            add 'parameter [0:0] LatchRequestn = 1'b0, WaitForGn(0) = 1'b1;'
```

add' reg [0:0] visual_Startn_current, visual_Startn_next;
add' reg [13*8-1:0] statename_n;

Every initiator will issue a request to a round robin arbiter for access to the rBus. The request signal is high for one clock tick only. Therefore if the request is not granted it must be latched so the arbiter knows the initiator still wishes to access the rBus.

Once a grant is received the latched request is taken away. At some time later the initiator may request access to the rBus again.

The arbiter can be parameterised to have 1 to N initiators requesting access to the same rBus. Each half duplex initiator can usually issue two separate requests One for a write and one for a read command. In this example N= 2 and both initiators are half duplex. For each write request the request and grant will be assigned a low number. The low numbers are 0 -> ((N/2) – 1). If N – 6 then the write request numbers are 0 to 2
For each read request the request and grant will be assigned a high number. The high numbers are (N/2) -> (N-1) If N – 4 then the read request numbers are 2 to 3
For (N = 0) -> (N – No_of_requestors – 1)

Create latch request code using template below replace 'N' with the specific requestor's number according to read or write
add' always @(rst or request_N or gnt_N or visual_Start_current)'
add'begin : Start_comb'
add'if (rst)'
add'begin'
add'latchReq_N<=0;'
add'visual_Start_next <= LatchRequestN;'
add'end'
add'else'
add'begin'
add'case (visual_Start_current)'
add'LatchRequestN'

```
add'begin'
add'statename <= "LatchRequestN";'
add'if ((request_N) && (gnt_N == 0))'
add'begin'
add'latchReq_N<=1;'
add'visual_Start_next <= WaitForGntN;'
add'end'
add'else'
add'begin'
add'visual_Start_next <= LatchRequestN;'
add'end'
add'end'
add'WaitForGntN:'
add'begin'
add'statename <= "WaitForGntN";'
add'if (gnt_N)'
add'begin'
add'latchReq_N<=0;'
add'visual_Start_next <= LatchRequestN;'
add'end'
add'else'
add'begin'
add'visual_Start_next <= WaitForGntN;'
add'end'
add'end'
add'default:'
add'begin'
add'visual_Start_next <= LatchRequestN;'
add'end'
add'endcase'
add'end'
add'end'
```

add`always @(posedge brClk)`
add`begin : Start`
add`if (rst)`
add`begin`
add`visual_Start_current <= LatchRequestN;`
add`end`
add`else`
add`begin`
add`visual_Start_current <= visual_Start_next;`
add`end`
add`end`
End ofLatch Requests template code

The algorithm below creates the grant and select functionality for the round robbin arbitrator.

Where H should be replaced by the read requestor number and L should be replaced by the write requestor number.
Detailed verilog code is not shown here .

For each (Request_L/ Request_H)
Check (Last Request)
if (last Request -= Request_L)
Check (Request_H == 1)
If True (Check Space Available to accept data)
If True (Check that we need to check for done)
If True (check Done — 1) // rBus is ready to accept a new request
If True (assert Grant_H)
If False Wait for Done and when true (assert Grant_H)
If False Assert (Grant_H)
If False wait for space available and loop back to (check that we need to check for done)
If False Check (Request_L — 1)

If True (Check that we need to check for done)

If True (check Done == 1) // rBus is ready to accept a new request.

If True (assert Grant_L)

If False Wait for Done and when true (assert Grant_L)

If False Assert (Grant_L)

End

Arbitration Logic, wrapper logic, a Top Level instantiation file and a 'Get Interface Signals' may be created in similar manner.

'Divide-by' Clocks

Algorithms for generation of any 'divide-by' clock to be used in the architecture and algorithms for the generation of Strobe, ClrStrobe and Sample signals may be as follows.

Algorithm for CLK Edge Identification

A, B = divide-by numbers.

```
if (A%2 == 0) || (B%2 == 0)
        NO_OF_STATES = LCM of A & B
else
        NO_OF_STATES = LCM of (A & B) * 2

NO_OF_EDGES = (NO_OF_STATES) /A

POS_EDGE = array of size NO_OF_EDGES
NEG_EDGE = array of size NO_OF_EDGES
```

Inialising/Choosing Clock Divide Functions for a CLK. CLK_TYPE_A

//chooses which CLK equation CLK_TYPE_A belongs to, based on whether A & A/2 are even numbers. Also

//chooses logic CLK. if LOGIC flag is high.

```
if A%2 = 0
  {
   if A/2 % 2 = 0                          // A is an even number
     CLK_TYPE_A = EVEN_EVEN                // A/2 is an even number
   else
     CLK_TYPE_A = EVEN_ODD// A/2 is an odd number
  }
else                                       // A is an odd number
  {
  if (A-1)/2 % 2 = 0
    {
    CLK_TYPE_A = ODD_EVEN                  // The number below A would be an even-
even CLK
     if LOGICA                             // If Logic Flag is high
       {
       CLK_TYPE_AL = ODD_EVEN_L// Create Logic CLK of type ODD_EVEN_L
       }
     else
       {
       CLK_TYPE_AL = NULL
       }
   else
    {
    CLK_TYPE_A = ODD_ODD // The number below A would be an even-odd CLK
    if LOGICA
      {
```

```
        CLK_TYPE_AL = ODD_ODD_L // Create Logic CLK of type ODD_ODD_L
      }
    else
      {
      CLK_TYPE_AL = NULL          // Do not create logic CLK
      }
  }
```

EVEN to EVEN CLKs

```
// Creates 2 arrays detailing the SYSCLK edges which have
POSEDGEs/NEGEDGEs

for (n=0) -> (n = NO_OF_EDGES - 1)
    POSEDGE[n] = n.A + 1

for (n=1) -> (n = NO_OF_EDGES)
    NEGEDGE[n-1] = A.(2n-1) / 2
```

EVEN to ODD

```
for (n=0) -> (n = NO_OF_EDGES - 1)
    POSEDGE[n] = n.A + 1

for (n=1) -> (n = NO_OF_EDGES)

    NEGEDGE[n-1] = A.(n+1)/2
```

ODD to EVEN

```
for (n=0) -> (n = NO_OF_EDGES - 1)
    if (n%2 = 0)
      POSEDGE[n] = A.n + 1
    Else
      POSEDGE[n-1] = A.n
```

```
for (n=1) -> (n = NO_OF_EDGES - 1)
   if (n%2 = 0)
     NEGEDGE[n-1] = [A.(2n-1) + 1] / 2
   else
     NEGEDGE[n-1] = [A.(2n-1)-1] / 2
```

ODD to ODD

```
for (n=0) -> (n = NO_OF_EDGES - 1)
    if (n%2 = 0)
      POSEDGE[n] = A.n + 1
    Else
      POSEDGE[n-1] = A.n
```

```
for (n=1) -> (n = NO_OF_EDGES - 1)
   if (n%2 = 0)
     NEGEDGE[n-1] = [A.(2n-1) - 1] / 2
   else
     NEGEDGE[n-1] = [A.(2n-1) + 1] / 2
```

ODD to ODD Logic Clock

```
for (n=0) -> (n = NO_OF_EDGES - 1)
   POSEDGE[n] = A.n + 1

for (n=1) -> (n = NO_OF_EDGES)
   if (n%2 = 0)
     NEGEDGE[n-1] = [A.(2n - 1) + 1] / 2
   else
     NEGEDGE[n-1] = [A.(2n - 1) + 3] / 2
```

ODD to EVEN Logic Clock

```
for (n=0) -> (n = NO_OF_EDGES - 1)
   POSEDGE[n] = A.n + 1

for (n=1) -> (n = NO_OF_EDGES)
   if (n%2 = 0)
     NEGEDGE[n-1] = [A.(2n - 1) + 3] / 2
   else
```

NEGEDGE[n-1] = [A.(2n - 1) + 1] / 2

Algorithm for CLK Generation

```
// Generates the CLK pulses based on the numbers associated with the POSEDGE and
NEGEDGE // arrays
// For hand-designed state machines, there is sufficient information in the above blocks
to
// generate the state table outputs for these clocks.

CLKA[0] = 0
PREV_CLK = 0

for (y = 0) -> (y = NO_OF_EDGES - 1)
    {
    for (n = 1) -> (n = NO_OF_STATES)
        {
        if (POSEDGE[y] == n)
            CLKA[n] = 1
        else if (NEGEDGE[y] == n)
            CLKA[n] = 0
        else
            CLKA[n] = PREV_CLK
        PREV_CLK = CLKA[n]
        }
    }
```

Algorithm for Generation of Strobe Signals

Strobe Signal

```
// Generates the Strobe signal based on the rule: 1st (fast) POSEDGE after (slow) NEGEDGE.

PREV_STROBE = 0          // value of Strobe signal in previous state
PREV_A = 0               // value of CLKA in previous state
PREV_B = 0               // value of CLKB in previous state

for (n = 0) -> (n = NO_OF_STATES - 1)
  {
  CURR_A = CLKA[n]       // assigns CURR_A the current value of CLKA
  CURR_B = CLKB[n]       // assigns CURR_B the current value of CLKB

  if (PREV_A == 0) && (CURR_A == 1) && (PREV_STROBE == 1)
      STROBED = 1        // data has been strobed on this edge

  if (PREV_B == 1) && (CURR_B == 0)
      STROBE[n] = 1      // sets Strobe on NEGEDGE of CLKB
  else if (PREV_A == 1) && (CURR_A == 0) && (STROBED)
      STROBE[n] = 0      // Clears Strobe on NEGEDGE of CLKA
  else
      STROBE[n] = PREV_STROBE // otherwise sets Strobe to its previous value

PREV_A = CLKA[n]         // sets PREV_A to current CLKA value
PREV_B = CLKB[n]         // sets PREV_B to current CLKB value
PREV_STROBE = STROBE[n]  // sets PREV_STROBE to current STROBE value
```

Strobe Signal (fast Logic CLK)

```
// generates the Strobe signal when the faster block has a logic CLK.
// variation on the rule for I/f -> I/f CLKs:
// POSEDGE (fast Logic CLK) before 1st NEGEDGE (fast Logic CLK) after
 NEGEDGE (slow I/F
// CLK).

PREV_STROBE = 0                    // value of STROBE in previous state
PREV_AL = 0                        // value of CLKAL in previous state
PREV_B = 0                         // value of CLKB in previous state

B_NEG = 0                          // stores a value related to a CLKB NEGEDGE

NEXT_EDGE_AL = 0                   // flag that controls the strobing edge of A

for (n = 0) -> (n = NO_OF_STATES-1)
    {
    STROBE[n] = 0   // STROBE set to 0 for every value of n (unless overwritten)

    CURR_AL = CLKAL[n]             // current value of AL
    CURR_B = CLKB[n]               // current value of B

if (PREV_AL = 0) && (CURR_AL = 1)
  POS_VALID_AL = n                 // set POS_VALID_AL on CLKAL POSEDGE

else if (PREV_AL = 1) && (CURR_AL = 0)
      POS_VALID_AL = 0             // reset POS_VALID_AL on CLKAL NEGEDGE

if (PREV_B = 1) && (CURR_B = 0)
```

```
        }
        if (POS_VALID_AL != 0)
          {
          STROBE[POS_VALID_AL-1] = 1 // if B NEGEDGE, and POS_VALID_A
isn't zero,
          STROBE[POS_VALID_AL] = 1   // overwrite the values for Strobe at the two
indexes
          B_NEG = n                  // given here, and set B_NEG to n.
          }
        else if (POS_VALID_AL = 0)
          {
          STROBE[n] = 1              // if POS_VALID_AL has been set to 0, set
          NEXT_EDGE_AL = 1           // NEXT_EDGE_AL and B_NEG
          B_NEG = n
          }
      }

    if (PREV_AL = 0) && (CURR_AL = 1) &&(NEXT_EDGE_A = 1)
          {                          // if POSEDGE CLKAL, and NEXT_EDGE is 1
          for (i = B_NEG) -> (i = n) // keep STROBE high from the value of
B_NEG to
            STROBE[i] = 1            // the current state

        NEXT_EDGE_AL = 0             // reset all flags
        POS_VALID_AL = 0
        B_NEG = 0
        }
      PREV_AL = CLKAL[n]             // set previous CLKs to current CLKs
      PREV_B = CLKB[n]
      }
```

Algorithm for Generation of ClrStrobe Signal

```
// generates the ClrStrobe signal
// This signal is asserted two CLK ticks before a slow CLK NEGEDGE, and de-asserted
on the
// NEGEDGE itself.
// ClrStrobe is used to override the Lstrobe internal signal, preventing a node from
Strobing data a clock
// tick before the NEGEDGE of the slower block with which it is communicating.

PREV_CLK_B = 0
CURR_CLK_B = 0

for (n = 0) -> (n – NO_OF_STATES – 1)
        {
        CLKA[n] = 0
        if (PREV_CLK_B — 1) && (CURR_CLK_B == 0)
                {
                CLKA[n-2] – 1
                CLKA[n-1] = 1
                }
        PREV_CLK_B – CURR_CLK_B
        }
```

Algorithm for Generation of Sample Signal

Sample Signal

// produces the Sample signal based on the rule:

// (fast) POSEDGE before 1st (fast) NEGEDGE after (slow) POSEDGE.

```
PREV_A = 0              // value of CLKA in previous state
PREV_B = 0              // value of CLKB in previous state
NEXT_EDGE_A = 0         // flag used in identification of correct sampling edge

for (n = 0) -> (n = NO_OF_STATES-1)
    {
    SAMPLE[n] = 0               // sample signal set low in every state
    CURR_A = CLKA[n]            // assigns CURR_A the current value of CLKA
    CURR_B = CLKB[n]            // assigns CURR_B the current value of CLKB

    if (PREV_A = 0) && (CURR_A = 1)
      POS_VALID_A = n    // records the state in which a valid posedge of A occurs

    if (PREV_B = 0) && (CURR_B = 1)
      {
      if (POS_VALID_A = 1)
        {
        SAMPLE [POS_VALID_A - 1] = 1  // overwrites previously stored values of
sample
        SAMPLE [POS_VALID_A] = 1          // based on POS_VALID signal
        POS_VALID_A = 0
        }
      else if (POS_VALID_A = 0)
            NEXT_EDGE_A = 1   // data will be sampled on next POSEDGE of A

    if (PREV_A = 0) && (CURR_A = 1) && (NEXT_EDGE_A = 1)
        {
        SAMPLE[n-1] = 1             // sets sample signal high for 2 ticks if
        SAMPLE[n] = 1               // NEXT_EDGE flag is set
        NEXT_EDGE_A = 0
        }
    PREV_A = CLKA[n]                // sets PREV_A to current CLKA value
```

```
    PREV_B – CLKB[n]                    // sets PREV_B to current CLKB value
}
```

Sample Signal (slow Logic CLK)

```
// generates the Sample signal when the slower block has a logic CLK.
// variation on the rule for i/f -> i/f CLKs:
// POSEDGE (fast CLK) before 1st NEGEDGE (fast CLK) after NEGEDGE (slow Logic
 CLK).

// SAMPLE is set low on every n, and may be overwritten by the process described here:
// Checks for POSEDGE of fast CLK. Stores VALID, and the SYSCLK number
associated with it.
// Clears VALID on a NEGEDGE. Checks for slow Logic NEGEDGE. If VALID is
high, then
// SAMPLE pulses high prior to the SYSCLK associated with VALID. If VALID is low,
then a
// NEXT_EDGE signal is set high, causing SAMPLE to pulse prior to the next fast
POSEDGE.

PREV_SAMPLE = 0                     // sets variables to 0
PREV_A = 0
PREV_BL = 0

NEXT_EDGE_A = 0

for (n = 0) -> (n = NO_OF_STATES-1)
    {
    SAMPLE[n] = 0   // SAMPLE signal set to zero for every n, unless overwritten
    CURR_A – CLKA[n]
    CURR_BL = CLKBL[n]

    if (PREV_A = 0 ) && (CURR_A = 1)   // stores value for POSEDGE of CLKA
```

```
        POS_VALID_A = n
    else if (PREV_A = 1) && (CURR_A = 0)        // reset value if NEGEDGE of CLKA
            POS_VALID_A = 0

    if (PREV_BL = 1) && (CURR_BL = 0)           // if NEGEDGE of CLKBL
      {
      if (POS_VALID_A != 0)                     // if POS_VALID_A is set, overwrite the
        {                                       // value of SAMPLE at the two indexes given
        SAMPLE [POS_VALID_A - 1] = 1            // & reset POS_VALID_A
        SAMPLE [POS_VALID_A] = 0
        POS_VALID_A = 0
        }
      else if (POS_VALID_A = 0)     // if POS_VALID_A is not set, then set
NEXT_EDGE
            NEXT_EDGE_A = 1

    if (PREV_A = 0) && (CURR_A = 1) &&(NEXT_EDGE_A = 1)
            {
            SAMPLE[n-1] = 1         // if POSEDGE A, and NEXT_EDGE_A is set
            SAMPLE[n] = 1           // overwrite previous SAMPLE value and set
            NEXT_EDGE_A = 0         // current value to 1. Reset the NEXT_EDGE
variable
            }
    PREV_A = CURR_A[n]              // set the PREV variables to CURRENT values
    PREV_BL = CURR_BL[n]
    }
```

Example. Divide-by-2, Divide-by-5 and Divide-by-6

// This example shows the necessary states and signals for a Divide-by-2 block communicating

// with Divide-by-5 and Divide-by-6 blocks.

// This state machine will require 30 states (LCM of 3 numbers)

Figures 6 and 7 illustrate the progression of states and the timing diagram for the corresponding state machine.

Generation of Interconnect Logic

Figure 8 shows two interconnect hierarchies and so how the same set of cores selected from a library of cores can be connected in radically different ways. Product teams decide on the functional logic required in a new ASIC (what needs to be selected from the library).

The following pseudo-code describes the top-level steps used to automatically generate the Interconnect Logic. New elements or functionality may be added to the interconnect in the future (E.g. Scan chain logic or some form of packet processing in the interconnect hierarchy) The top-level design will allow new elements to be easily added.

```
CLK_BLK[..] = array of Clock Generator objects of size NO_CLK_BLK.
REG_BLK[..] = array of Register Bridge objects of size NO_REG_BLK.
ARB_BLK[..] = array of Arbitration objects of size NO_ARB_BLK
IPWRAPPER[..] = array of IP Core Wrapper objects of size
                NO_IPWRAPPER_BLK
VALID = boolean value used to decide if the interconnect hierarchy that you want
        to create Logic for if valid.

VALID = Validate Interconnect Hierarchy( )
IF (VALID == 0)
    Exit
For (n=0) -> (n = NO_CLK_BLK-1)
    Create Clock Logic( CLK_BLK[n] )
    Add to Instantiation File ( CLK_BLK[n] )
For (n=0) -> (n = NO_REG_BLK-1)
```

Create Bridge Logic (REG_BLK[n])

Add to Instantiation File (REG_BLK[n])

For (n=0) -> (n – NO_ARB_BLK-1)

Create Arbitration Logic (ARB_BLK[n])

Add to Instantiation File (ARB_BLK[n])

For (n=0) -> (n – NO_IPWRAPPER_BLK-1)

Create IP Wrapper Logic (IPWRAPPER[N])

Add to Instantiation File (IPWRAPPER[N])

Validate Interconnect Hierarchy

The Interconnect Hierarchy is validated before any verilog is generated. This checks if any architectural assumptions, interconnection rules or clock generation rules are broken. This will automatically enforce certain rules as a designer inputs information (i.e. parameter value ranges, connections between blocks). The following is a non exhaustive list of the checks that can only be performed once the diagrams are complete and the user wishes to generate verilog

1. Each rBus path has at least one rBus target interface connected to it or stated another way each Register Bridge has at least one core connected to it in the Register Diagram.
2. There are no more than 64 cores in total (limited by size of Source Address).
3. Only processor cores can address the Register Bridges.
4. There is only one unique path from an mBus Initiator to an mBus Target or stated another way there are no loops in the diagram. All paths start with an Initiator and end with a Target.
5. All blocks in the diagram are connected to a Clock Generation Block
6. If a Clock Generator is used as a parent then the following two conditions must hold (a) none of the blocks below it in the interconnect hierarchy talk directly to any other blocks at a higher level and if (b) all blocks below it in the interconnect hierarchy can have their Clock_Frequency derived from it.
7. The memory map has been correctly defined, there are no overlapping areas and that the reserved address has not been assigned (Initial boot address of the boot processor).

The validation stage will also generate warnings. It will be possible to change the severity of a warning so that they will stop the generation of verilog. The following is a non-exhaustive list of these warnings

1. Any parameters that are still set to there default value.
2. All unused interfaces within a wrapper (mBus or rBus interfaces).
3. The Required Bandwidth for an Arbitration block is greater than it's Output Bandwidth (freq * bus width)
4. The Required Bandwidth is greater than the Output Bandwidth on any mBus Initiator port
5. The sum of the bandwidth's allocated is greater that the Memory Bandwidth of an mBus Target.

Create Clock Logic

The following pseudo-code describes the high level steps used to create the logic for a Clock Generator Block. The parameters used in the creation of Clock Logic are fully described previously.

NAME = Unique name for this Clock Generator Block
CLK_FREQ = Clock frequency generated by block.
PARENT_CLK = Parent Clock used to derive the generated Clock frequency
IS_LOGIC_CLK = boolean value which specifies if a Logic Clock should be generated or not.
CLK_SIGNALS[ ] = array of objects from the blocks connected to the Clock Generator of size CONNECTIONS. Holds information such as divide by ratios etc

Clock Edge Identification (CLK_RATIOS[..]) (see section 4 of sdg000.009)
For (n=0) -> (n = CONNECTIONS-1)
    Choose CLK divide Function (CLK_RATIO[n],IS_LOGIC_CLK)(section 4.1 sdg000.009)
Create the Clock State Machine
For (n=0) -> (n = CONNECTIONS-1)

Clk Generation Algorithm (CLK_SIGNALS[n]) (section 5 sdg000.009)

Strobe Signal Algorithm (CLK_SIGNALS[n]) (section 6 sdg000.009)

Sample Signals Algorithm (CLK_SIGNALS[n]) (section 8 sdg000.009)

Create Clock Out Interface (n)

The high level clock functions are shown in Figure 9.

1. Clock State Machine. All the verilog associated with this function will be generated by the algorithms described previously. The parameters that are used as input to this algorithm are Configurable Parameters and Diagramming Rules. The size and complexity of the state machine is dependant on the three parameters described namely – Parent Clock, Is_Logic_Block and Divide_By_Ratio.
2. Clock Out Interface. There will be one interface per upper level interconnect block (i.e. arb. wrapper. bridge) that the Clock Generation block is connected to. This interface will drive the necessary Clock signals (Sample, Strobe etc) to the connected block. The verilog for this function will be created from a standard template that will be instantiated within the block the required number of times. The signal names for each interface will be changed to ensure that they are unique. The width of the output Clock Bus within this template code will be parameterizable and will depend on the number of sample and strobe signals that need to be driven into the connected block. This interface will also contain logic to allow the Clock Out Interface to be switched off by an interconnect block in order to save power.

Create Bridge Logic

The following pseudo-code describes the high level steps used to create the logic for a Register Bridge. The parameters used in the creation of logic for a Register Bridge are fully described previously

NAME = Unique name for this Register Bridge

CLK_SIGNALS = Reference to an object which describes the Clock Signals for this Block (see section 4)

BASE_ADDR = The Base address for the block

MBUS_PORTS [..] = array of connected mBus Port objects of size NO_MBUS_PORTS

RBUS_WIDTH = the width of the rBus.

RBUS_TARGETS[..] = array of connected rBus Target objects of size NO_RBUS_TARGETS.

```
Create Clock Interface (CLK_SIGNALS)
For (n=0) -> (n = NO_MBUS_PORTS-1)
        mBus Target Logic (MBUS_PORTS [n] )

Round Robin Arbitrator ( NO_MBUS_PORTS)

mBus to rBus state machine ( )
 rBus Initiator Port ( )
Clock Interface( )
For (n=0) -> (n = NO_RBUS_TARGETS-1)
        rBus Select Logic ( RBUS_TARGETS[n] )
```

The high level Register Bridge functions are shown in Figure 10.

1. mBus Target Interface The verilog for this function will be created from a standard template that will be instantiated within the block the required number of times. The signal names for each interface will be changed to ensure that they are unique The mBus Target interface accepts an mBus read and write request. It stalls the interface until the request is handled (read response or write acknowledgement). It waits for an access grant from the arbitrator and passes the request to the rBus Initiator interface and handles the response
2. Round Robin Arbitrator. The verilog for this function will be created from a standard template. The template will be configured with a parameter defining the number of mBus Target Interfaces that must be arbitrated. The Round Robin arbitrator polls each mBus Target interface for rBus read or write requests in a cyclic manner each time the rBus is idle. It grants access to the rBus for the first request it finds at any mBus Target Interface

3 rBus Initiator Interface. The verilog for this function will be created from a standard template. The signal names will be changed to ensure that they are unique for each register bridge created The rBus Initiator interface translates the mBus to rBus request and visa versa in the opposite direction. The address offset of the register you wish to access is passed to each of the Select Line Drivers.

4. Select Line Driver. The verilog for this function will be created from a standard template. The signal names will be changed to ensure that they are unique. The same template will be used to create the select line logic in the bridge, wrapper and arbitrator blocks. The function will be parameterised with the range of addresses that it recognises. The Select Line Driver looks at each address offset passed to it and decides if the select line should be driven high.

5. Select Lines. The top level Register Bridge block is parameterised to define the number of select lines supported. The value of the parameter is equal to the number of rBus Targets connected to the rBus.

6. Clock Interface. The verilog for this function will be created from a standard template. The block's external signal names will be changed to ensure that they are unique The same template will be used in all the interconnect blocks (i.e. arb. wrapper, bridge). The Clock Interface distributes the clock signal to all functions within the block. It will route the necessary Sample and Strobe signals to the mBus and rBus interfaces defined for this interconnect block.

Create Arbitration Logic

The following pseudo-code describes the high level steps used to create the Logic for the upward path Arbitration Block. The parameters used in the creation of logic for an Arbitrator are fully described previously.

NAME = Unique name for this Arbitration Block

CLK_SIGNALS – Reference to an object which describes the Clock Signals for this Block (see section 4)

MBUS_IN_PORTS |..| = array of connected mBus Port objects and Input Port parameters of size NO_MBUS_PORTS

MBUS_OUT_PORT= mBus objects which describes the mBus Output port parameters

MBUS_TARGETS= number of mBus targets.

Create Upward Path Logic

Create Clock Interface (CLK_SIGNALS)

For (n=0) -> (n = NO_MBUS_PORTS-1)

mBus Input Logic (MBUS_IN_PORTS [n])

create FIFO logic (MBUS_IN_PORTS [n])

Create Arbitrator (NO_MBUS_PORTS , MBUS_IN_PORTS[..])

mBus Output Port (MBUS_OUT_PORT)

For (n=0) -> (n = MBUS_TARGETS-1)

mBus Select Logic (MBUS_TARGETS)

The mBus and rBus are point to point bi-directional buses that can operate in full or half duplex mode In this when you draw an Arbitration block and connect an mBus to one of its ports you are inferring an upward and downward path along the bus. At the level of abstraction described in this document it is normally sufficient to show this as one path. Arbitration blocks must store multiple read and write requests on the upward path (from an mBus Initiator) and multiple read responses and write acknowledgements on the downward path (from an mBus Target). This split is shown in the two diagrams below. Although I have drawn two separate bubbles to make the diagrams more readable they would in all probability be part of the same high level interconnect block. For this reason I have only shown a Clock and Register Interface in the one of the bubbles.

The high level upward path Arbitration functions are shown in Figure 11.

1. mBus Input Interface. The verilog for this function will be created from a standard template that will be instantiated within the block the required number of times (the same template is used for the up / down paths). The signal names for each interface will be changed to ensure that they are unique. The mBus input interface clocks in mBus read and write requests on the correct edge and passes the data to the Fifo buffers.

2 FIFO. The verilog for this function will be created from a standard template that will be instantiated within the block the required number of times. The size of the buffers is defined by passing a parameter into the function when it is instantiated. The FIFO stores the data associated with the mBus requests. It will stall the mBus Input Interface when its full.

3. Arbitration. The verilog for this function will be created from a standard template The template will be configured with a parameter defining the number of mBus Input Interfaces that must be arbitrated and a bandwidth allocation parameter per interface that defines its arbitration priority. The arbitrator will grant access to the mBus Output port based on an arbitration algorithm.

4. mBus Output port The verilog for this function will be created from a standard template (the same template is used for the up / down paths) The signal names will be changed to ensure that they are unique. The mBus Output port will pass mBus requests to the next level in the interconnect. The address of the mBus target you wish to access is passed to each of the Select Line Drivers.

5. Select Line Driver. The verilog for this function will be created from a standard template. The signal names will be changed to ensure that they are unique. The same template will be used to create the select line logic in the bridge. wrapper and urbitrator blocks. The function will be parameterised with the range of addresses that it recognises. The Select Line Driver looks at each address passed to it and decides if the select line should be driven high.

6. Select Lines. The top level Arbitration block is parameterised to define the number of select lines supported. The value of the parameter is equal to the number of times the mBus upward path is split (number of destinations - mBus Target / mBus Input ports).

7. Clock Interface. The verilog for this function will be created from a standard template. The block's external signal names will be changed to ensure that they are unique. The same template will be used in all the interconnect blocks (i.e. arb. wrapper. bridge). The Clock Interface distributes the clock signal to all functions within the block. It will route the necessary Sample and Strobe signals to the mBus and rBus interfaces defined for this interconnect block. It will turn the Clock signals off when programmed to do so via the register interface to save power consumption.

8. Register Target Interface. The verilog for this function will be created from a standard template that will be used only for creating Arbitration block Register Target interfaces. The block's external signal names will be changed to ensure that they are unique. It will allow access to the configurable registers within the block..
9. Read-back Throttle. Verilog signal. Its name will have to be unique if the two blocks are separated otherwise it will simply be an internal signal within the block..

The following pseudo-code describes the high level steps used to create the Logic for a downward path Arbitration Block.

```
NAME – Unique name for this Arbitration Block
CLK_SIGNALS = Reference to an object which describes the Clock Signals for
this Block (see section 4)
MBUS_IN_PORTS [..] = array of connected mBus Port objects and Input Port
parameters of size          NO_MBUS_PORTS
MBUS_OUT_PORT=  mBus objects which describes the  mBus Output port
parameters
MBUS_TARGETS= number of mBus targets
### Create Downward Path Logic ###
For (n=0) -> (n = NO_MBUS_PORTS-1)
        mBus Input Logic (MBUS_IN_PORTS [n] )
        create FIFO logic (MBUS_IN_PORTS [n] )

Create Readback Decode Arbitrator ( NO_MBUS_PORTS . MBUS_IN_PORTS [..])
Create Write Ack Arbitrator ( NO_MBUS_PORTS . MBUS_IN_PORTS [..])

mBus Output Port (MBUS_OUT_PORT )
For (n=0) -> (n = MBUS_TARGETS-1)
        mBus Select Logic ( MBUS_TARGETS)
```

The high level downward path Arbitration functions are shown in Figure 12

1. mBus Input Interface. The verilog for this function will be created from a standard template that will be instantiated within the block the required number of times (the same template is used for the up / down paths). The signal names for each interface

will be changed to ensure that they are unique. The mBus input interface clocks in mBus read and write requests on the correct edge and passes the data to the Fifo buffers.

2 FIFO. The verilog for this function will be created from a standard template that will be instantiated within the block the required number of times. The size of the buffers is defined by passing a parameter into the function when it is instantiated. The FIFO stores the data associated with the mBus responses A read-back throttle is sent to the upward path Arbitration block when the Fifo is full.

3 Read Back Decode. The verilog for this function will be created from a standard template. The template will be configured with a parameter defining the number of mBus Input Interfaces that must be arbitrated and a bandwidth allocation parameter per interface that defines its arbitration priority. The Read Back Decode is separated from the Write Back Acknowledge because they use separate wires in the mBus and therefore arbitration for the mBus can be performed independently of each other.

4. Write Back Acknowledge. This functionality provided here is very similar to the Read Back Decode except the Write Acknowledgements are arbitrated. The verilog template used will be very similar to that used for the Read Back Decode.

5. mBus Output port. The verilog for this function will be created from a standard template (the same template is used for the up / down paths). The signal names will be changed to ensure that they are unique. The mBus Output port will pass mBus responses to the next level in the interconnect. The Source Identifier of the mBus Initiator Interface that generated the original request is passed to each of the Select Line Drivers.

6. Select Line Driver. The verilog for this function will be created from a standard template. The signal names will be changed to ensure that they are unique. The same template will be used to create the select line logic in the bridge. wrapper and arbitrator blocks. The function will be parameterised with the range of Source Identifiers that it recognises. The Select Line Driver looks at each Source Identifier passed to it and decides if the select line should be driven high.

7. Select Lines. The top level Arbitration block is parameterised to define the number of select lines supported. The value of the parameter is equal to the number of times the

corresponding mBus downward path is split (number of destinations ~ mBus Initiators / mBus Input ports)

Figure 13 shows the mBus up and down paths. The upward path is shown by solid lines and the downward path is shown by dashed lines. The downward path arbitration functionality is shown as a square where there is only one input and output port. The functions within the downward path will be exactly the same as the upward path. except in this case the Arbitration functionality within the read-back decode and write back acknowledgement functions is redundant. The Logic compiler can recognise this fact and remove the redundant verilog

Create Wrapper Logic

The following pseudo-code describes the high level steps used to create logic for a Core Wrapper Block. The parameters used in the creation of logic for a core are fully described previously

NAME = Unique name for this Arbitration Block
MBUS_TARGETS= number of mBus targets.
For (n=0) -> (n = MBUS_TARGETS-1)
mBus Select Logic (MBUS_TARGETS)

Preferably there are two core types that will be available in the library. viz. an mBus Initiator core (e.g Ethernet. PCI. USB) and an mBus Target core (e.g. SRAM memory. Flash Memory. Buffer Memory). Its unlikely that a core could have both an mBus Target and Initiator interface therefore these two core types are treated separately. All cores contained in a library of cores will need a wrapper similar to the ones described here. Each core will have its own unique requirements: therefore the wrapper will vary somewhat from core to core.

The high level core (mBus Initiator) wrapper functions are shown in Figure 14.

1. Core Logic This is handcrafted logic that is unique to each core. The tool will not modify this logic
2. DMA Engine Handcrafted logic that is unique to the core. The tool will not modify this logic
3. rBus Target Interface. Handcrafted logic that is unique to each core (logic in each core will be very similar). The tool will not modify this logic.
4. mBus Initiator Interface. Handcrafted logic that is unique to each core (logic in each core will be very similar). The tool will not modify this logic.
5. Select Line Driver. The verilog for this function will be created from a standard template The signal names will be changed to ensure that they are unique. The same template will be used to create the select line logic in the bridge, wrapper and arbitrator blocks. The function will be parameterised with the range of memory addresses that it recognises The Select Line Driver looks at each address passed to it and decides if the select line should be driven high.
6. Select Lines. The top level wrapper block is parameterised to define the number of select lines supported. The value of the parameter is equal to the number of times the corresponding mBus upward path is split (number of destinations – mBus Target / mBus Input ports).

All core wrapper blocks will be designed with an rBus interface and one or more mBus interfaces. In addition a single mBus can be split to multiple destinations using select lines. The cores will then be incorporated into the library and can be used in multiple designs. In a design it maybe decided that a particular interface is not needed (i.e. only communicate with a UART block over the rBus). The compiler will automatically handle the circumstances where signals need to be tied off.

Add to Instantiation File

The interconnect logic generated will be completely flat. All blocks will be instantiated at the same level. One top-level instantiation file will be created Each block within the interconnect will be listed in the file. The top-level input and output signals will be

extracted from each of the interconnect blocks and declared in the top-level instantiation file. The following information will be extracted for each signal:

- Signal Name
- Signal Width
- Signal Direction
- Is it an External Signal (Pin out)
- Value to tie an Input signal to if it is unused.

The global parameters described previously will be declared and passed into each of the interconnect blocks. The appendix section contains an example verilog module and how that module would be declared at a higher level.

Appendix 1: Sample Verilog

The following is some example verilog showing the type of verilog that will generate for the Clock State Machine.

```
module strobe_generation ( clk, RESET, clk2, strobe[TARGET_TOTAL-1:0],
                                        clrStrobe[TARGET_TOTAL-1:0]
                                        sample[TARGET_TOTAL-1:0]);
parameter TARGET_TOTAL = 2; // Parameters
input parent_clk
parameter [2:0] State0 = 29'b0; State1 = 29'b10; State2 = 29'b100;
reg [2:0]       current_state, next_state;

always @(RESET)
begin  Start0_comb
       strobe_next <= strobe;
       clrStrobe_next <= clrStrobe;
       sample_next <= sample;

       if (RESET == 1)
```

```
                begin strobe = 0; sample – 0; clrStrobe – 0; next_state = State1; end
        else begin
                case (current_state)
                        State0:
                        begin clk2 <= 0; strobe <= 2'b00; clrStrobe <= 2'b10; sample <=
2'b11; next_state <– State1. end
                        State1.
                        begin clk2 <= 1; strobe <= 2'b00; clrStrobe <– 2'b10; sample <=
2'b11; next_state <= State2; end
                        State2:
                        begin clk2 <= 0; strobe <= 2'b10; clrStrobe <= 2'b01; sample <=
2'b00; next_state <= State3; end
                        default:
                        begin next_state <= State0; end
                endcase
        end
end
end module
```

The following shows an example of a verilog module and a top level instantiation file.

```
module foo (in_sig, clk, rst, out_sig).
        input in_sig;  wire in_sig;
        input clk; wire clk.
        input rst; wire rst;
        output out_sig; reg out_sig;

        always @ (posedge clk)
        begin
                if(rst == 1)
                        out_sig <= 0;
                else
                        out_sig <= in_sig;
        end
endmodule
```

Example Module

```
module foo_test_top:
        reg SystemClock :
        wire in_sig :
        wire clk :
        wire rst :
        wire out_sig :

        foo dut( .in_sig ( in_sig ), .clk ( clk ), .rst ( rst ), .out_sig ( out_sig )
endmodule
```

CLAIMS

1 A program tool for the generation of a large scale integrated circuit, said tool comprising the steps of:

(a) defining an architecture comprising a central shared memory and a multiplicity of data handling cores.

(b) defining interconnect logic separately from said cores, said interconnect logic defining data paths that require all data exchanges between said cores to proceed by way of said shared memory.

2. A program tool according to claim 1 wherein said interconnect logic prescribes a hierarchy of data aggregation whereby each core is coupled to said memory by way of at least one level of arbitration and at least some cores are coupled to said memory by way of at least two levels of arbitration

3. A program tool according to claim 1 or claim 2 and wherein the tool provides the step of defining register paths separate from the data paths and proceeding between data processor cores and others of said cores.

4. A program tool according to any foregoing claim wherein the tool further comprises the steps of:

obtaining said cores from a library of cores; and

defining parameters for each of said cores.

Abstract

A program tool automatically generating interconnect logic for a system on a chip is based on a library of operational cores and on a architecture which requires all data exchange between cores to proceed via shared memory, which may be 'off-chip'. The architecture includes a data aggregation technique for access to memory with successive levels of arbitration.

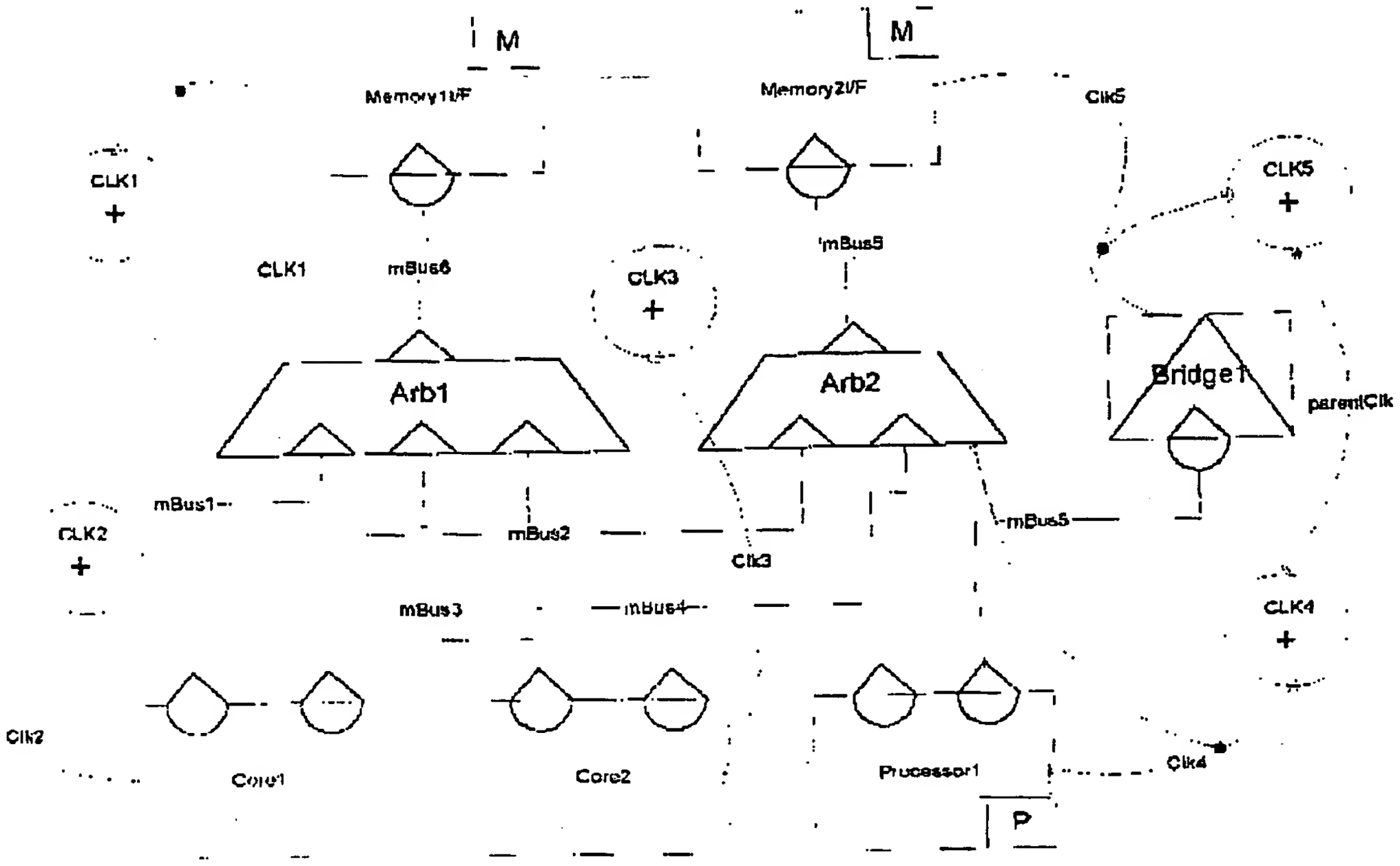
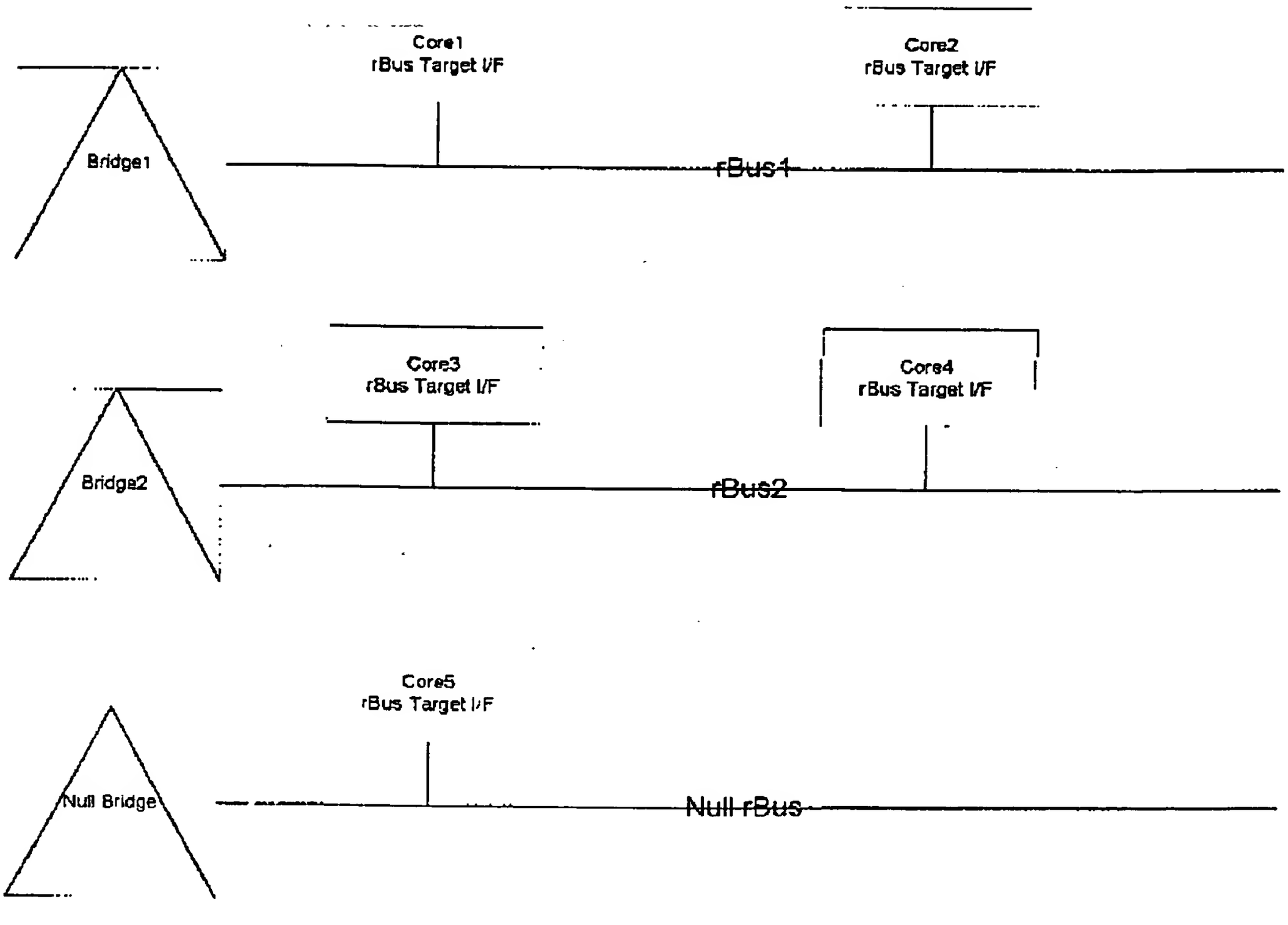

M
Memory1I/F
M
Memory2I/F
Clk5
CLK5
+
CLK1
+
CLK1
mBus6
CLK3
+
mBus5
Arb1
Arb2
Bridge1
parentClk
mBus1
CLK2
+
mBus2
Clk3
mBus5
mBus3
mBus4
CLK4
+
Clk2
Core1
Core2
Processor1
Clk4
P

FIG.1



Sample Register Path Diagram

FIG.2

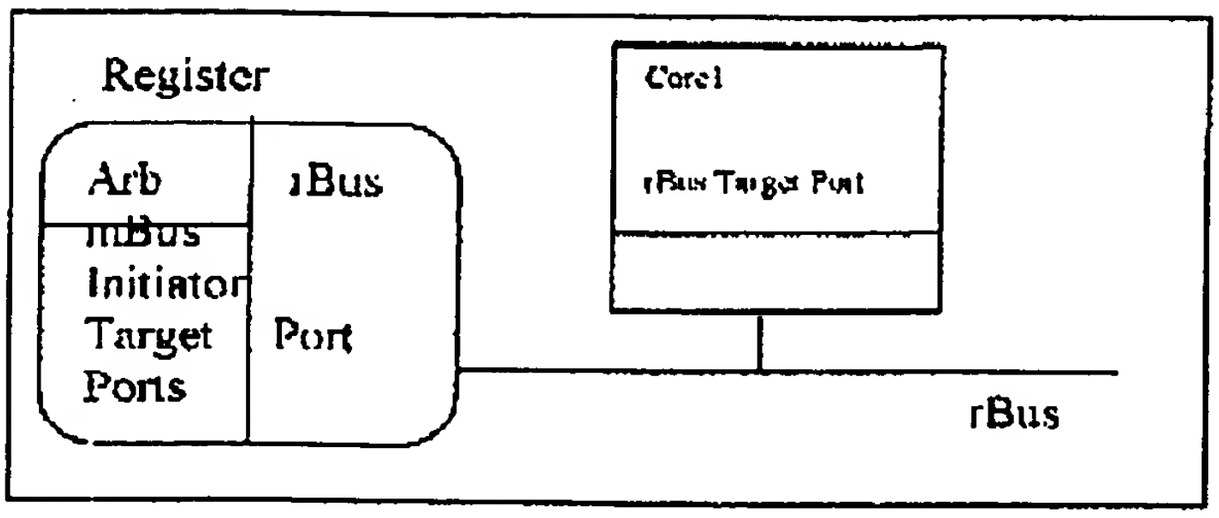

Register
Arb
Initiator
Target
Ports
Port
Core1
rBus Target Port
rBus

FIG.3

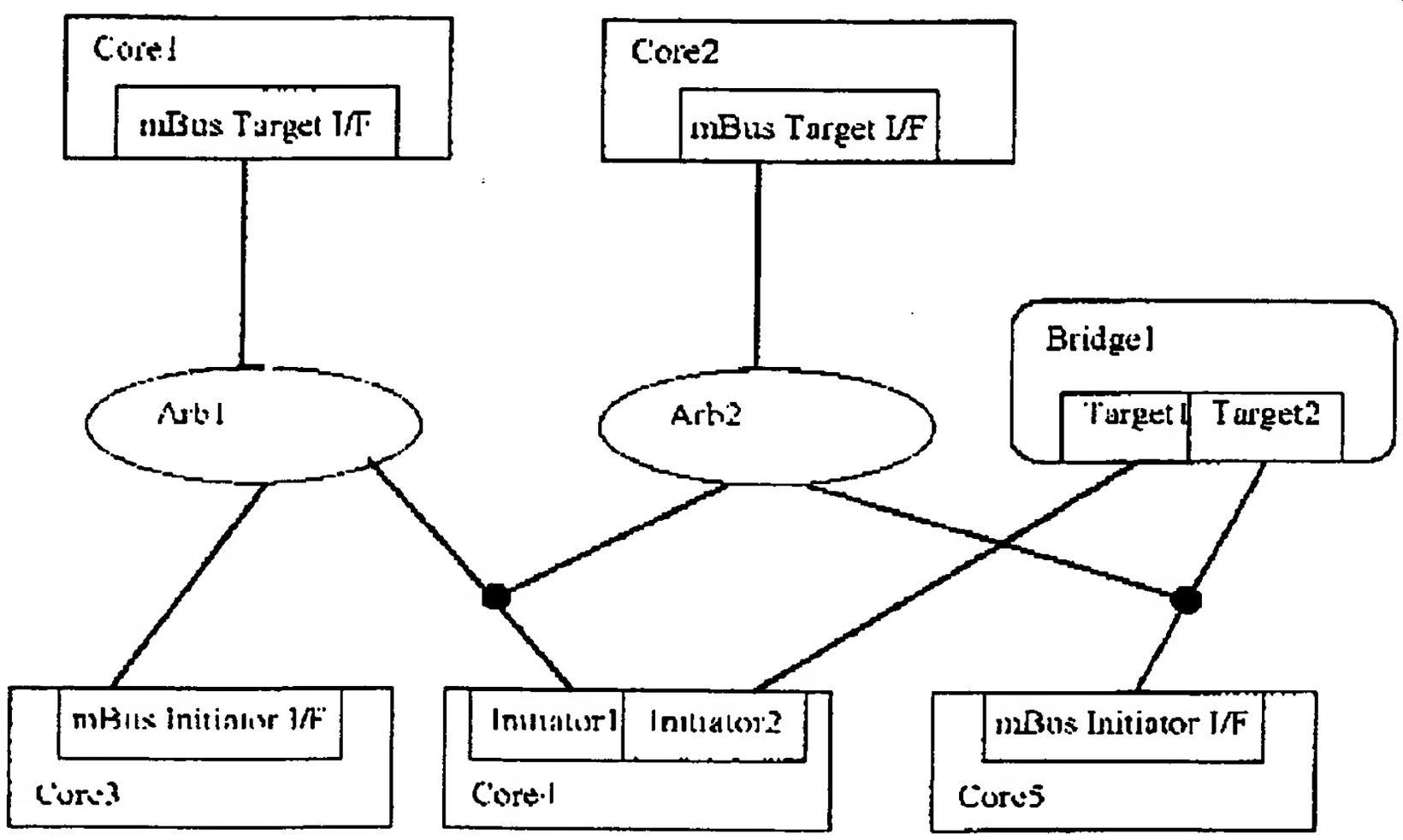

Core1
mBus Target I/F
Core2
mBus Target I/F
Bridge1
Target1
Target2
Arb1
Arb2
mBus Initiator I/F
Core3
Initiator1
Initiator2
Core4
mBus Initiator I/F
Core5

FIG.4

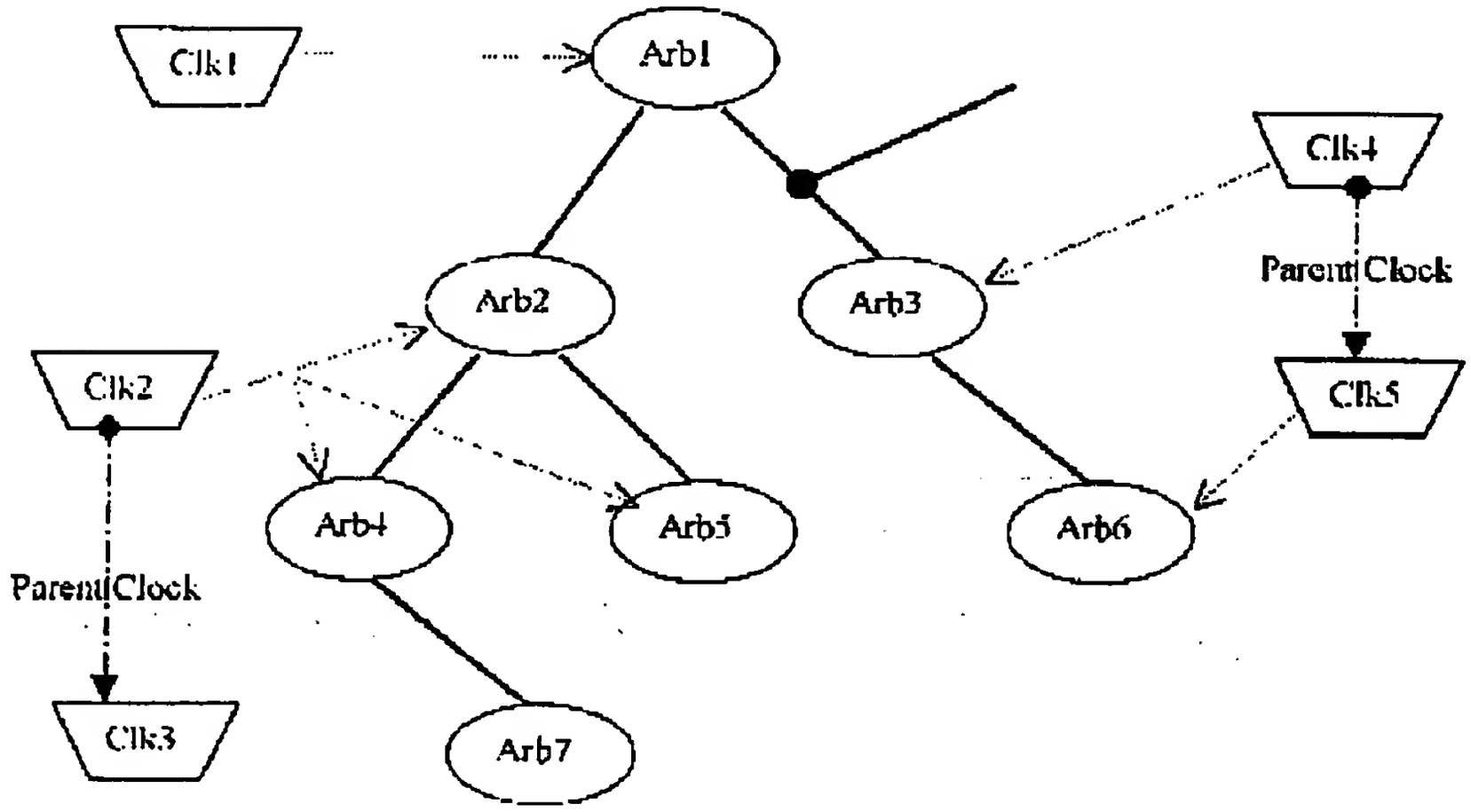

Clk1
Arb1
Clk4
Parent Clock
Arb2
Arb3
Clk2
Clk5
Arb4
Arb5
Arb6
Parent Clock
Clk3
Arb7

FIG.5

State	CLK2	CLK5	CLK6	Strobe5	ClrStrobe5	Sample5	Strobe6	ClrStrobe 6	Sample 6
0	0	0	0	0	1	1	0	0	1
1	1	1	1	0	1	1	0	0	1
2	0	0	1	1	0	0	0	1	0
3	1	0	1	1	0	0	0	1	0
4	0	0	0	0	0	1	1	0	0
5	1	1	0	0	0	1	1	0	0
6	0	1	0	0	1	0	0	0	1
7	1	1	1	0	1	0	0	0	1
8	0	0	1	1	0	0	0	1	0
9	1	0	1	1	0	0	0	1	0
10	0	0	0	0	1	1	1	0	0
11	1	1	0	0	1	1	1	0	0
12	0	0	0	1	0	0	0	0	1
13	1	0	1	1	0	0	0	0	1
14	0	0	1	0	0	1	0	1	0
15	1	1	1	0	0	1	0	1	0
16	0	1	0	0	1	0	1	0	0
17	1	1	0	0	1	0	1	0	0
18	0	0	0	1	0	0	0	0	1
19	1	0	1	1	0	0	0	0	1
20	0	0	1	0	1	1	0	1	0
21	1	1	1	0	1	1	0	1	0
22	0	0	0	1	0	0	1	0	0
23	1	0	0	1	0	0	1	0	0
24	0	0	0	0	0	1	0	0	1
25	1	1	1	0	0	1	0	0	1
26	0	1	1	0	1	0	0	1	0
27	1	1	1	0	1	0	0	1	0
28	0	0	0	1	0	0	1	0	0
29	1	0	0	1	0	0	1	0	0

FIG.6

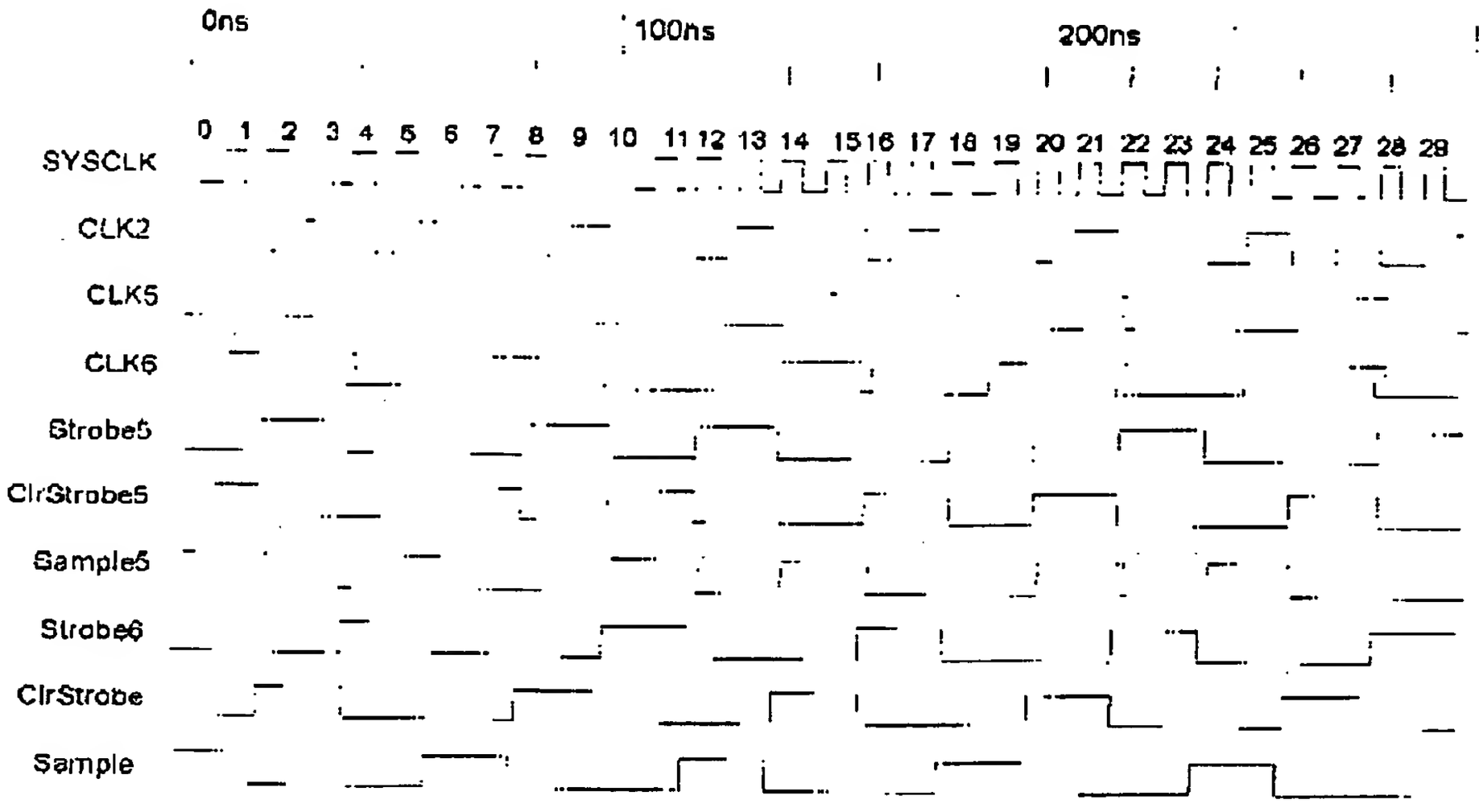

0ns
100ns
200ns
SYSCLK
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
CLK2
CLK5
CLK6
Strobe5
ClrStrobe5
Sample5
Strobe6
ClrStrobe
Sample

FIG.7

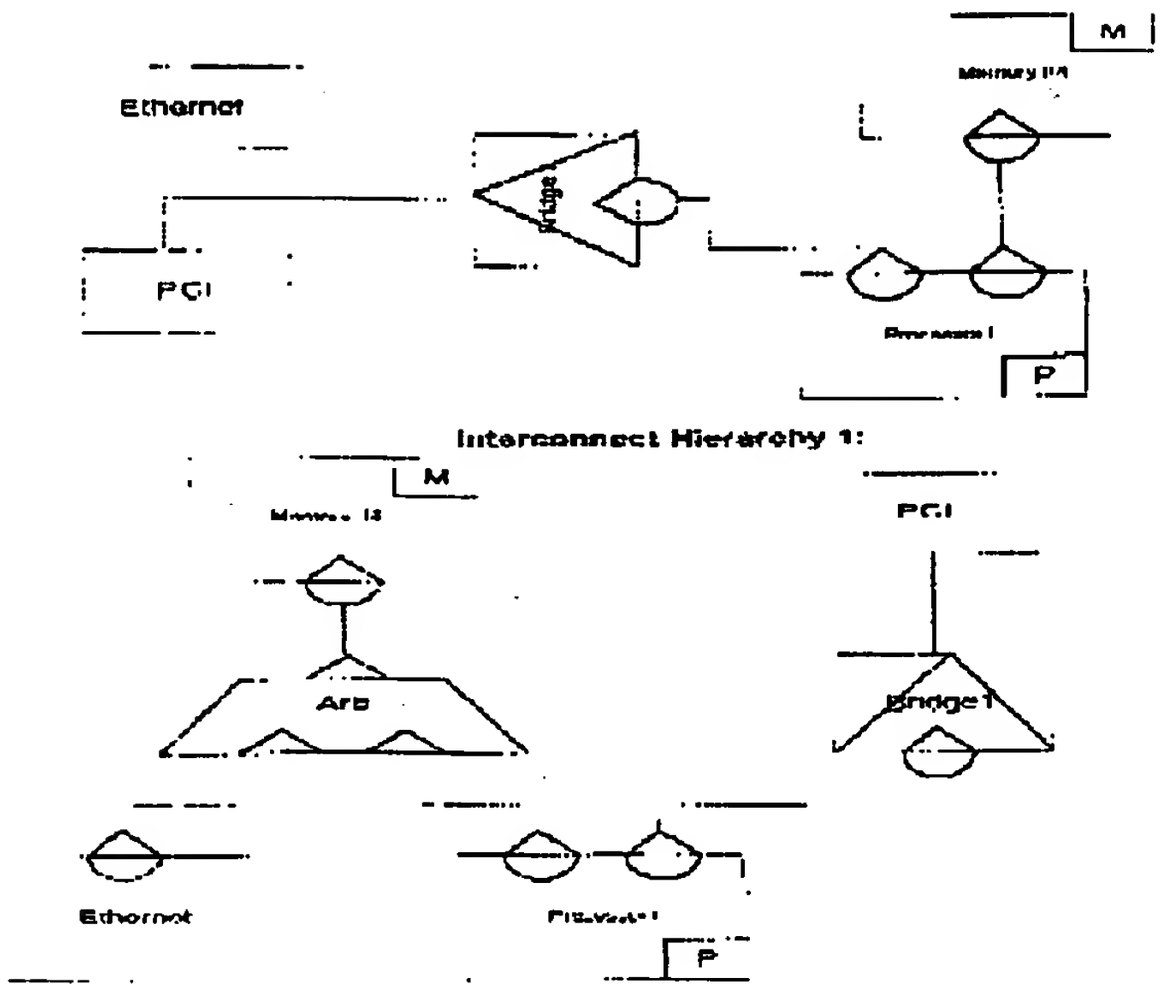

M
Ethernet
PCI
P
Interconnect Hierarchy 1:
M
PCI
Arb
Ethernet
P
Interconnect Hierarchy 2:

FIG.8

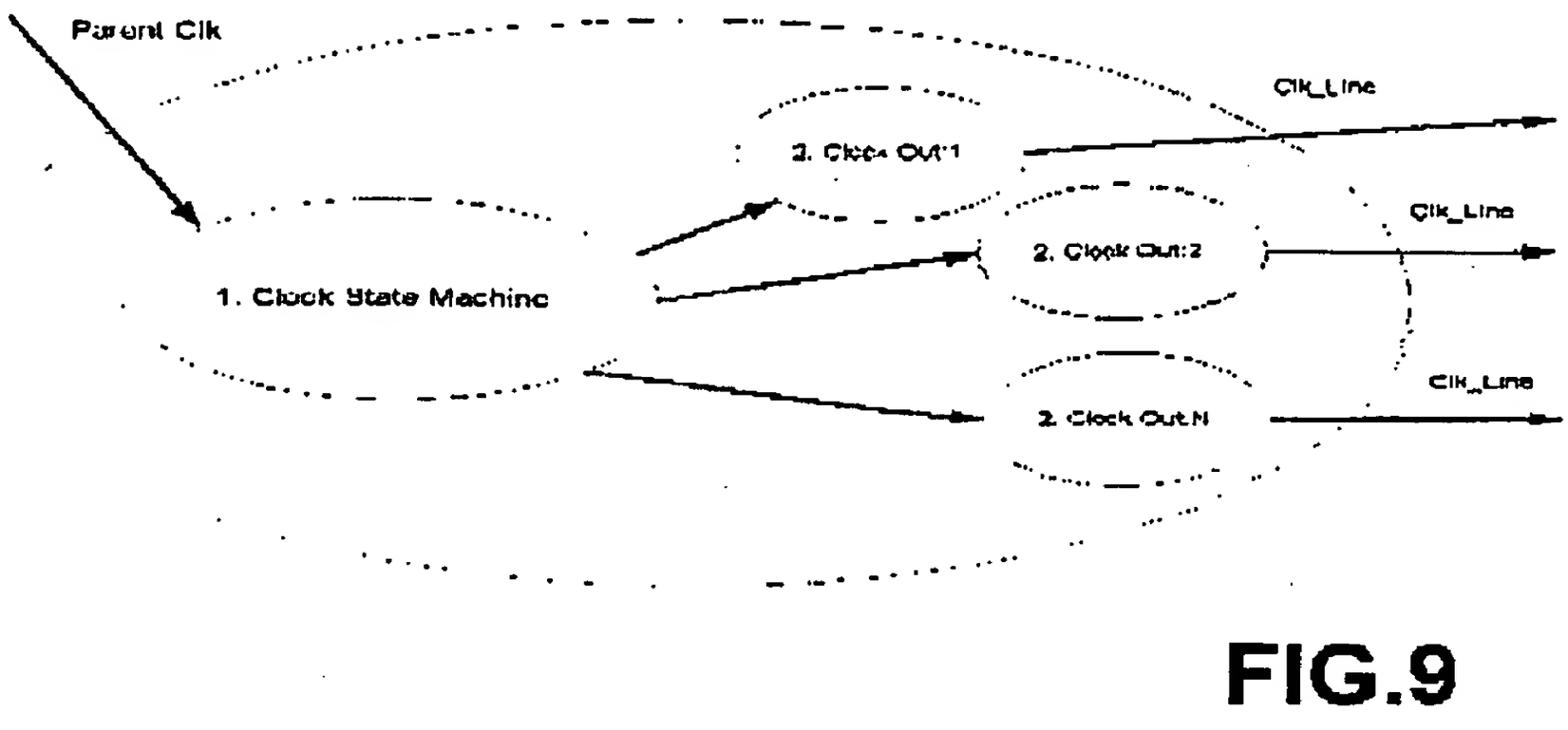

Parent Clk
1. Clock State Machine
2. Clock Out:1
2. Clock Out:2
2. Clock Out:N
Clk_Line
Clk_Line
Clk_Line

FIG.9

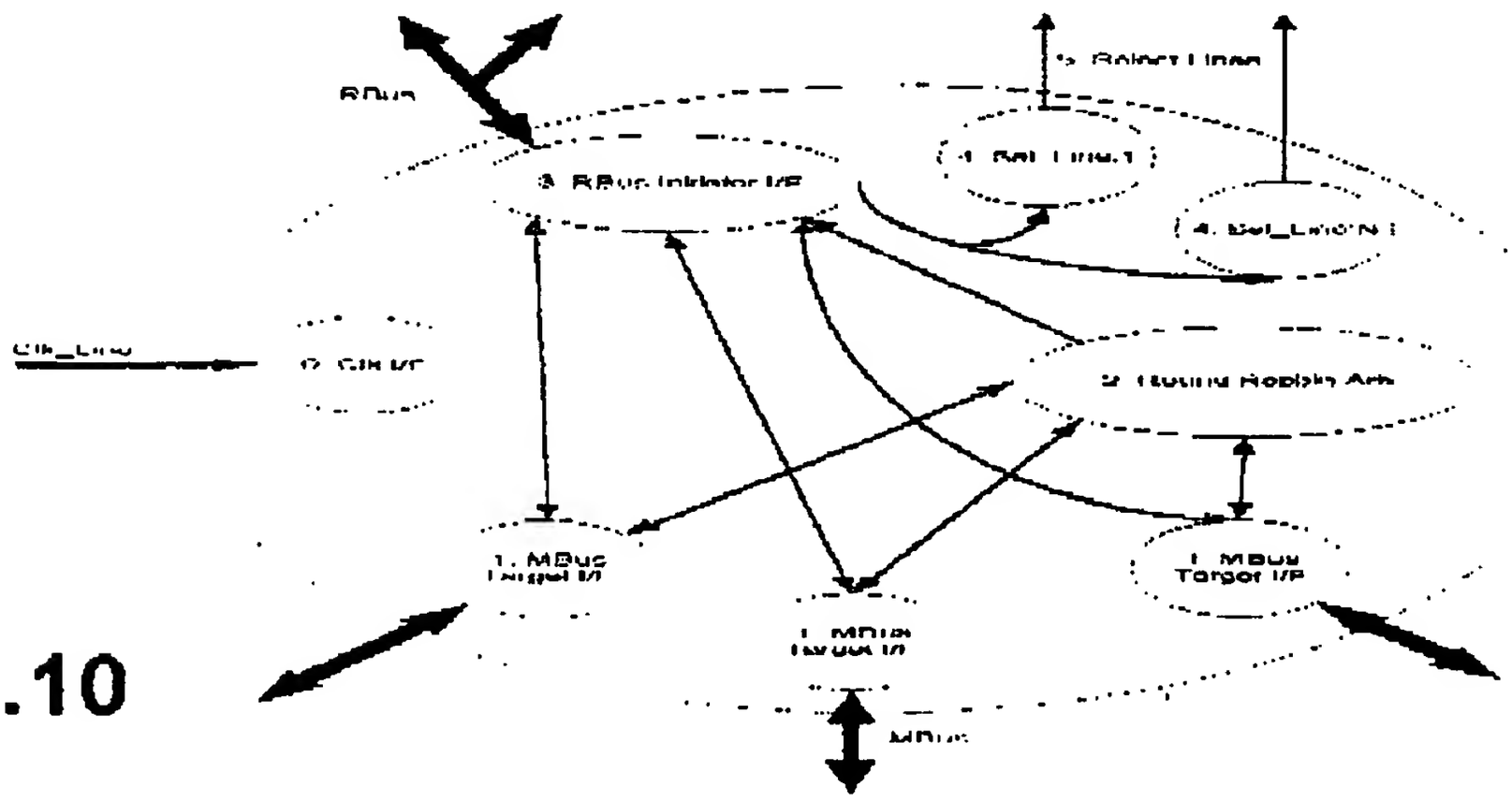


FIG.10

9/11

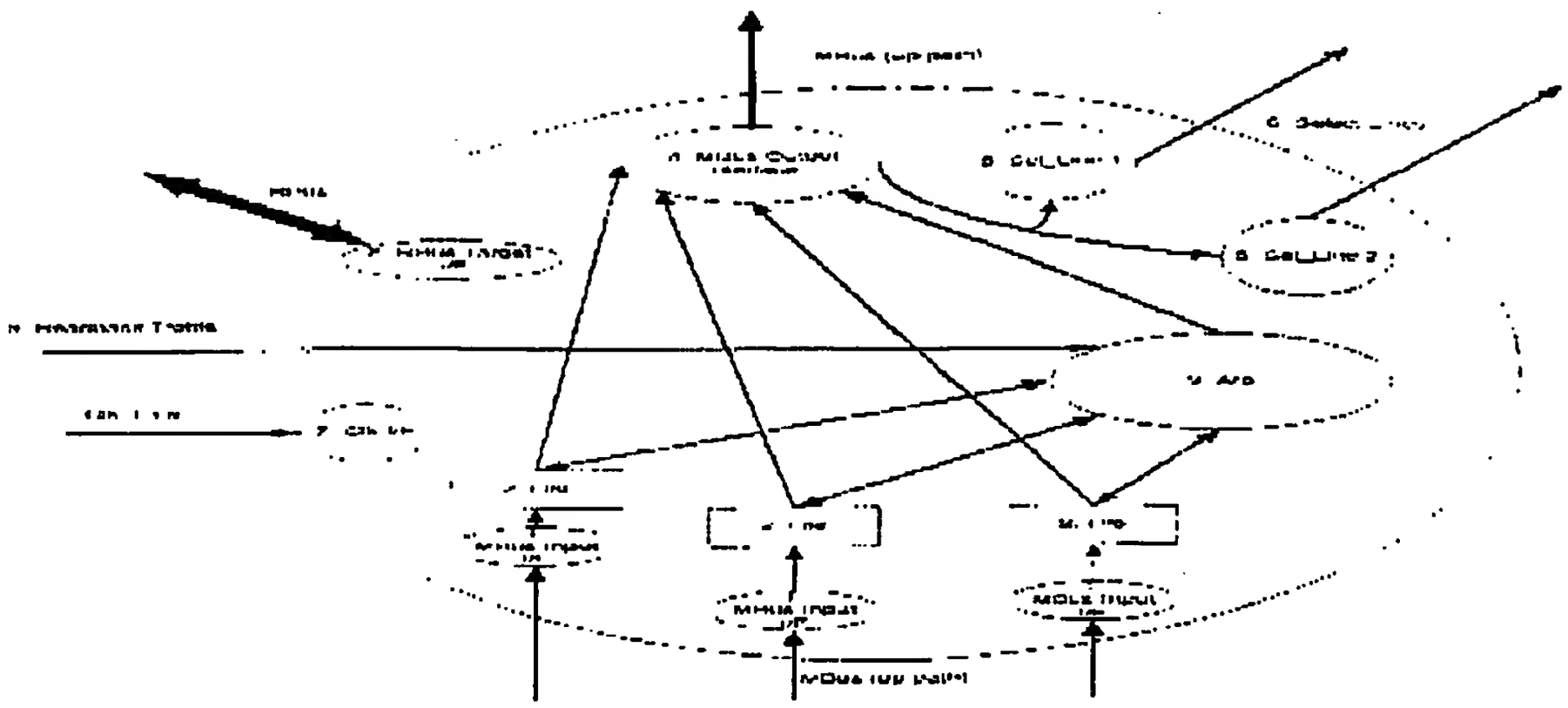


FIG.11

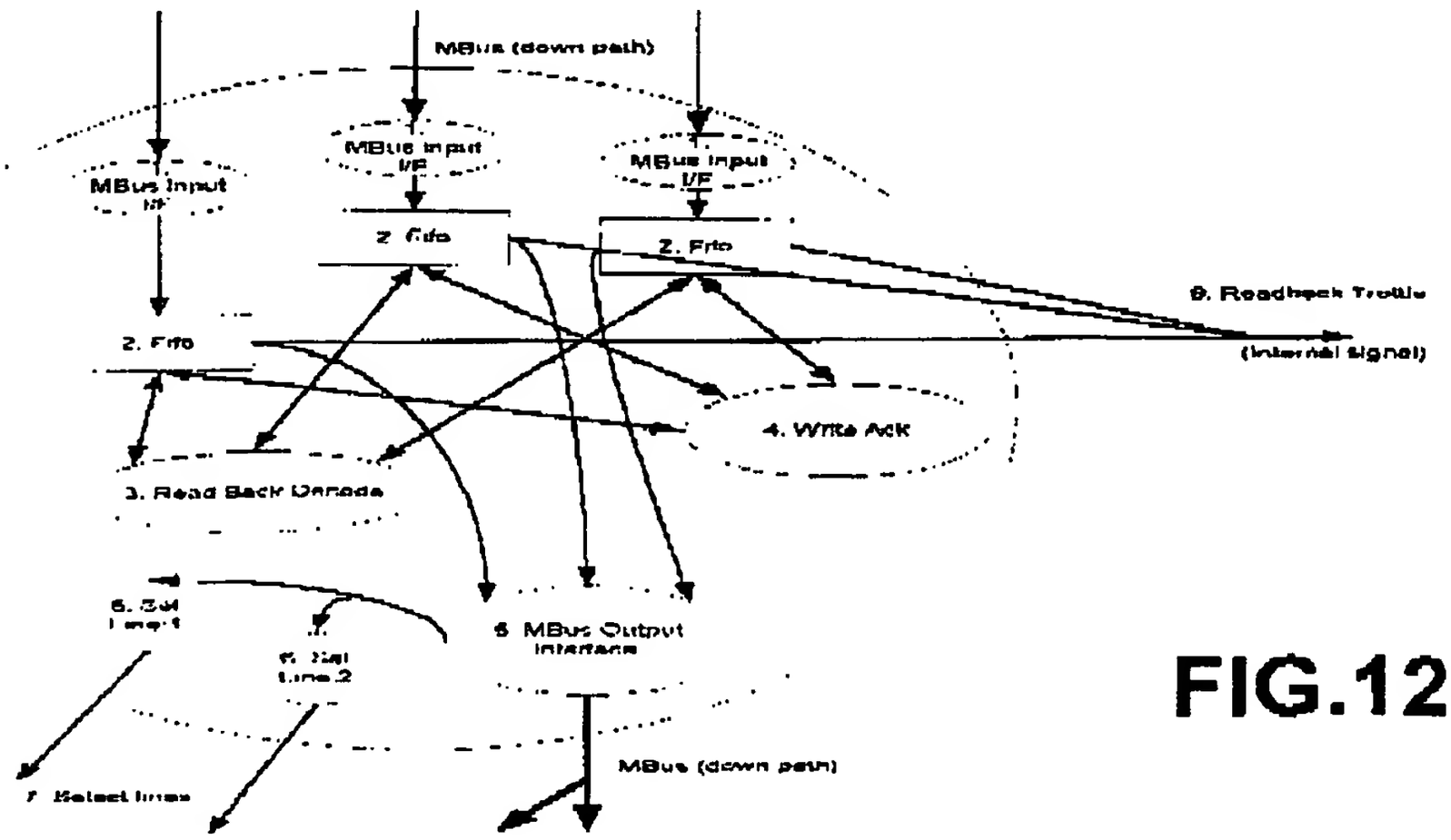


FIG.12

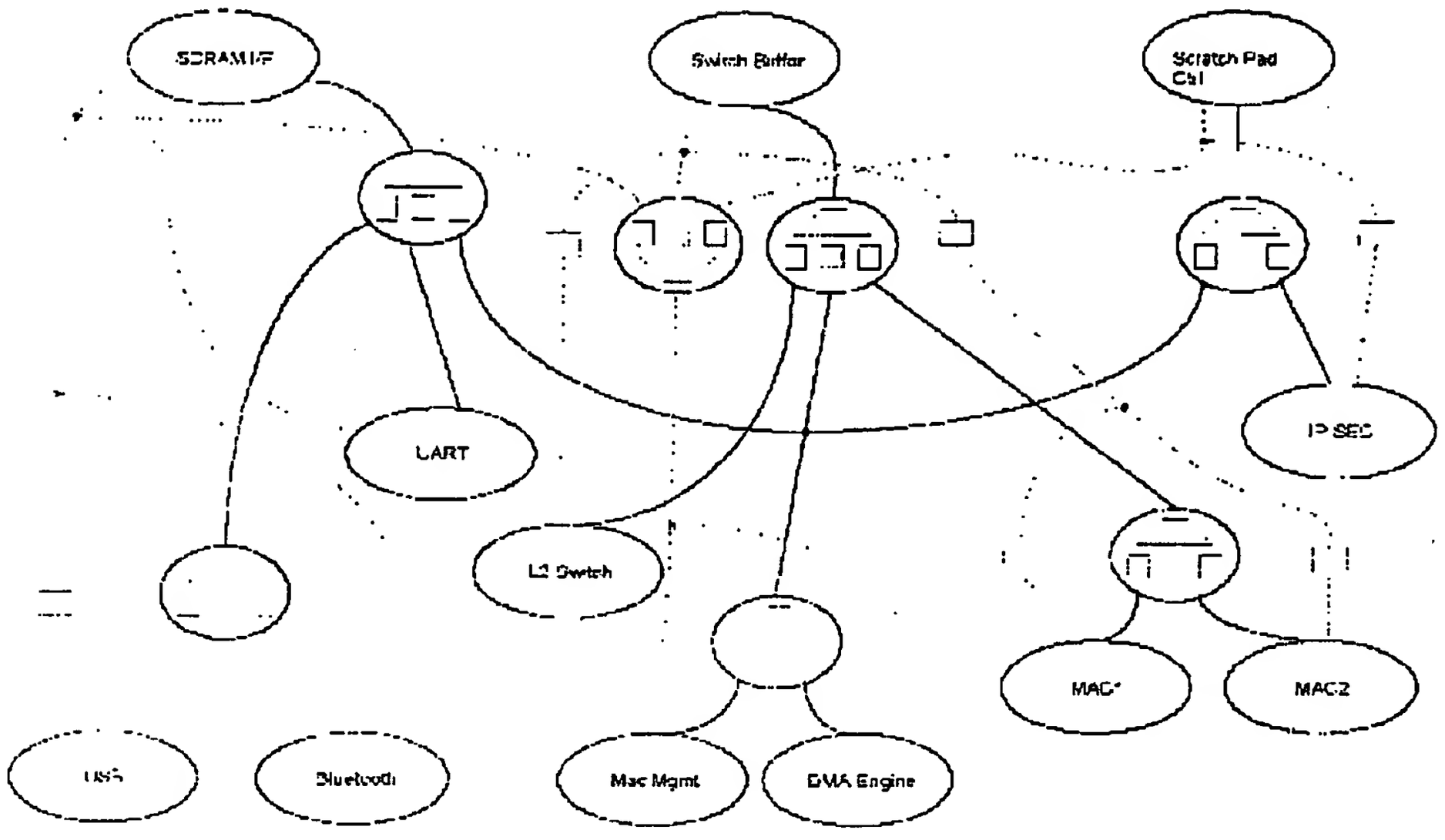

SDRAM I/F
Switch Buffer
Scratch Pad Ctl
UART
IP SEC
L2 Switch
MAC1
MAC2
USB
Bluetooth
Mac Mgmt
DMA Engine

FIG.13

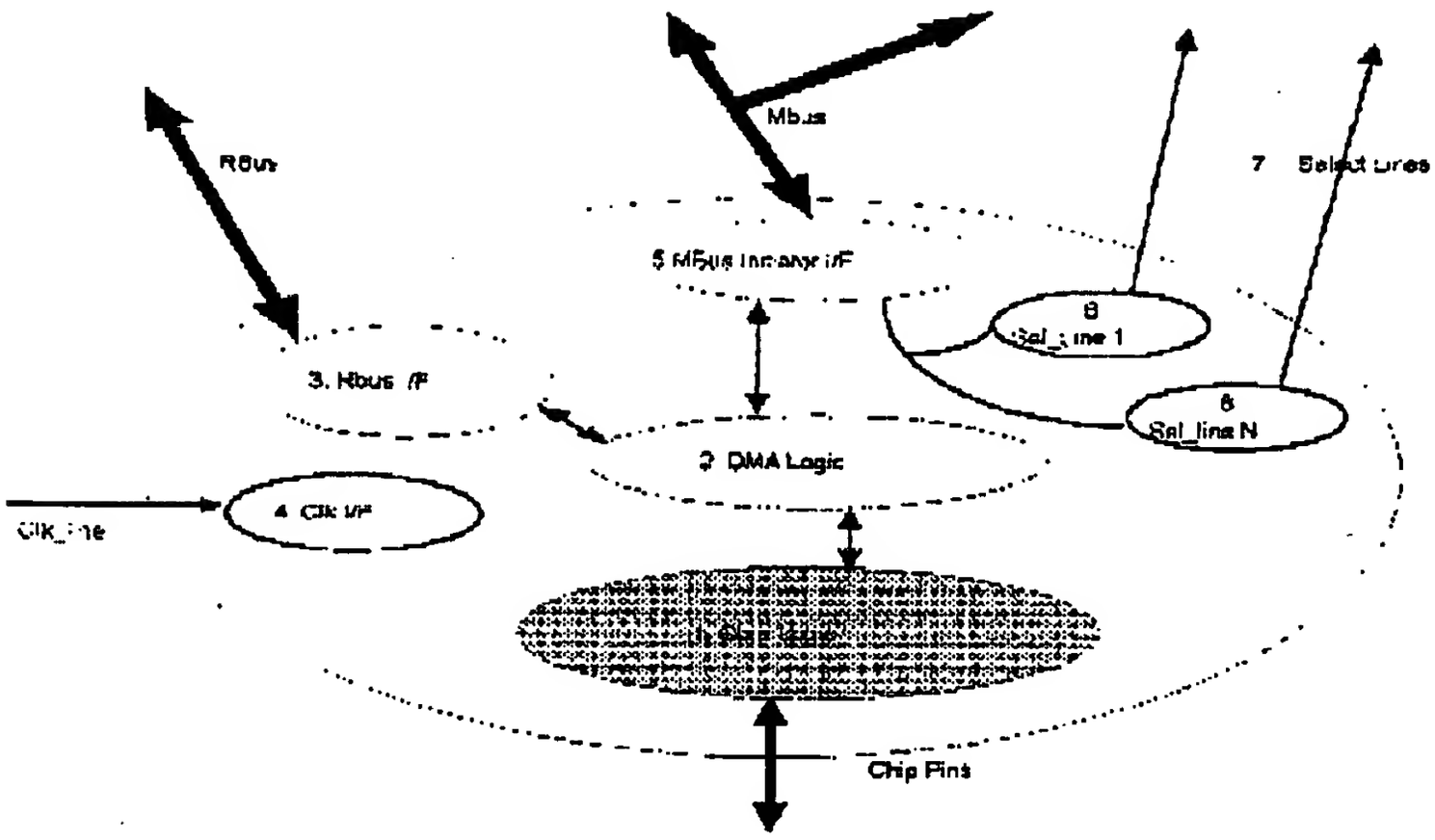

Mbus
Rbus
7 Select Lines
5 MBus Interface
8
Sel_line 1
3. Rbus IF
6
Sel_line N
2 DMA Logic
4 Clk I/F
CLK_File
Chip Pins

FIG.14